Titre: Title:	Preuves de non réalisabilité et filtrage de domaines pour les problèmes de satisfaction de contraintes : application à la confection d'horaires
Auteur: Author:	Sandrine Paroz
Date:	2009
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Paroz, S. (2009). Preuves de non réalisabilité et filtrage de domaines pour les problèmes de satisfaction de contraintes : application à la confection d'horaires [Thèse de doctorat, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/8281/

UNIVERSITÉ DE MONTRÉAL

PREUVES DE NON RÉALISABILITÉ ET FILTRAGE DE DOMAINES POUR LES PROBLÈMES DE SATISFACTION DE CONTRAINTES : APPLICATION À LA CONFECTION D'HORAIRES

SANDRINE PAROZ

DÉPARTEMENT DE MATHÉMATIQUES ET DE GÉNIE INDUSTRIEL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION

DU DIPLÔME DE PHILOSOPHIÆ DOCTOR (Ph.D.)

(MATHÉMATIQUES DE L'INGÉNIEUR)

AVRIL 2009

Your file Votre référence
ISBN: 978-0-494-49423-3
Our file Notre référence
ISBN: 978-0-494-49423-3

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

PREUVES DE NON RÉALISABILITÉ ET FILTRAGE DE DOMAINES POUR LES PROBLÈMES DE SATISFACTION DE CONTRAINTES : APPLICATION À LA CONFECTION D'HORAIRES

présentée par : PAROZ Sandrine

en vue de l'obtention du diplôme de : Philosophiæ Doctor

a été dûment acceptée par le jury d'examen constitué de :

M. SOUMIS François, Ph.D., président

M. HERTZ Alain, Doct. ès Sc., membre et directeur de recherche

M. GALINIER Philippe, Doct., membre et codirecteur de recherche

M. GENDRON Bernard, Ph.D., membre

M. AMALDI Edoardo, Doct. ès Sc., membre externe

À mes parents

REMERCIEMENTS

Tout d'abord je tiens à remercier mon directeur Alain Hertz pour m'avoir encouragée à faire cette thèse, pour sa disponibilité à répondre à mes questions et mes doutes et pour son soutien financier. Je remercie aussi mon codirecteur Philippe Galinier pour son financement et pour ses remises en question qui m'ont permise d'avoir un autre point de vue sur certains sujets.

Merci aussi à mes collègues et ex-collègues de bureau, en particulier Christian pour son aide informatique et Rim pour nos petites discussions.

Un très grand merci à mes parents et à mon frère qui m'ont soutenue tout au long de mes études et qui m'ont encouragée à distance avec la même intensité que si j'étais présente auprès d'eux.

Finalement, un merci particulier à mon conjoint Marcel pour m'avoir aussi appuyée et motivée et surtout pour avoir supporté et apaisé toutes mes angoisses et mon stress de fin de thèse.

RÉSUMÉ

De nombreux problèmes, théoriques ou appliqués, de grande taille et très contraints n'ont aucune solution qui respecte toutes les contraintes. Nous allons travailler avec des problèmes de ce type-là et qui peuvent être modélisés comme des problèmes de satisfaction de contraintes (CSP). Nous allons particulièrement nous intéresser aux causes de la non réalisabilité. En effet, il est très intéressant et très utile de pouvoir comprendre la cause de la non réalisabilité afin de pouvoir modifier les problèmes initiaux pour les rendre réalisables.

Afin de faire cela, nous voulons pouvoir détecter des sous-problèmes de l'instance de départ, plus petits, non réalisables et irréductibles et qui permettent d'expliquer la non réalisabilité du problème original. Nous appelons de tels sous-ensembles des sous-ensembles incohérents irréductibles de contraintes ou de variables. Comme exemple théorique, nous allons étudier le problème de satisfaisabilité booléenne (problème SAT). À titre d'exemple pratique, nous traiterons un problème de fabrication d'horaires pour le personnel navigant aérien. En effet, dans ces cas-là, les gestionnaires qui fabriquent les horaires aimeraient pouvoir détecter la cause de la non réalisabilité des problèmes afin de pouvoir modifier certaines données pour obtenir un horaire réalisable.

D'un autre côté, pour les problèmes réalisables, il arrive fréquemment qu'une valeur du domaine d'une variable soit impossible dans le sens qu'il n'existe aucune solution vérifiant toutes les contraintes dans laquelle la variable a cette valeur. Dans de tels cas, nous voulons pouvoir supprimer ces valeurs impossibles des domaines des variables. Cette technique est appelée le filtrage des domaines.

Cette thèse a deux buts principaux. Le premier consiste à mettre en oeuvre des outils qui repèrent automatiquement les sous-ensembles irréalisables irréductibles de contraintes ou de variables pour des problèmes non réalisables. Ces algorithmes seront adaptés pour

deux sortes de problèmes: le problème de satisfaisabilité booléenne et un problème de confection d'horaires pour le personnel navigant aérien. Le deuxième but consiste à développer un algorithme de filtrage des domaines dans le cas de problèmes réalisables. Nous allons développer un tel algorithme de filtrage pour une contrainte globale de CSP bien définie: la contrainte SomeDifferent.

Comme les problèmes que nous traitons sont de grande taille et très contraints, nous utilisons des heuristiques de recherche tabou pour effectuer la recherche de sous-ensembles incohérents irréductibles ou pour supprimer les valeurs impossibles des problèmes originaux. Ensuite, nous utilisons des algorithmes exacts pour vérifier les résultats obtenus.

La première partie de la thèse traite de la recherche de sous-ensembles irréalisables irréductibles de contraintes et de variables pour le problème de satisfaisabilité booléenne communément appelé problème SAT.
La deuxième partie présente un algorithme de filtrage des domaines pour la contrainte SomeDifferent.
Finalement, dans la troisième partie, nous appliquons les outils de détection de sous-ensembles irréalisables de contraintes à des problèmes de confection d'horaires pour le personnel navigant aérien.

ABSTRACT

A significant number of large and very constrained problems, be it theoretical or applied, have no solution that satisfies all the constraints. The present work focuses on this kind of problems that can be modeled as constraint satisfaction problems (CSP). We will be particularly interested in investigating the causes of impossibility of a feasible solution. Explaining the causes of failure to achieve a feasible solution is in fact extremely important and useful; since it may allow us to reformulate the original problem and transform it into a problem with a feasible solution.

For that purpose, we wish to detect smaller, incoherent and irreducible subproblems of the considered infeasible problem. We will call such subsets infeasible irreducible subsets (IIS) of constraints or variables. As a theoretical example, we will study the boolean satisfiability problem. And as a practical example, we will consider a crew scheduling problem, for which managers in charge of building a schedule would like to be able to detect the reasons of an infeasibility, in order to modify some of the constraints and obtain a feasible schedule.

On the other hand, for some of the feasible problems, it also frequently happens that a value in the domain of a variable is impossible in the sense that there is no solution verifying all the constraints in which the variable has this value. In those cases, we want to be able to delete these impossible values from the domains of the variables. Such a technique is called domain filtering and will be studied for a specific global CSP constraint: the SomeDifferent constraint.

This thesis has two main goals. The first goal is to implement tools that automatically detect the infeasible irreducible subsets of constraints or variables for infeasible problems. Such algorithms will be adapted for two problems: the boolean satisfiability problem and a crew scheduling problem. The second goal is to develop a domain filter-

ing algorithm for feasible problems.

As the considered problems are very constrained and of large size, we will use tabu search heuristics in our algorithms to extract infeasible irreducible subsets of constraints or variables in infeasible problems, and to detect impossible values in the domains of the variables in feasible problems. Then, we will use exact algorithms to validate the results produced by our heuristic methods.

The first part of the thesis focuses on the search of infeasible irreducible subsets of constraints and variables for the boolean satisfiability problem commonly called SAT problem.

The second part of the thesis presents a domain filtering algorithm for the SomeDifferent constraint.

Finally, in the third part, the detection tools of infeasible irreducible subsets of constraints are applied to a crew scheduling problem.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ABBRÉVATIONS ET SYMBOLES

coMSS ..... Complémentaire d'un sous-ensemble satisfaisable maximal
CSP ..... Problème de satisfaction de contraintes
HS-C ..... Algorithme HittingSet en mode contraintes
HS-V ..... Algorithme HittingSet en mode variables
IS ..... Sous-ensemble incohérent
IIS ..... Sous-ensemble incohérent irréductible
IIS-C ..... Sous-ensemble incohérent irréductible de contraintes
IIS-V ..... Sous-ensemble incohérent irréductible de variables
LSCP ..... Problème de recouvrement de grands ensembles
MSS ..... Sous-ensemble satisfaisable maximal
MUSC ..... Sous-ensemble minimal incohérent de clauses (problème SAT)
MUSV ..... Sous-ensemble minimal incohérent de variables (problème SAT)
MCUS ..... Sous-ensemble incohérent de cardinalité minimum (problème SAT)
SAT ..... Problème de satisfaisabilité booléenne
USCP ..... Problème de recouvrement à coût unitaire

$\mathbb{N}$ ..... Ensemble des entiers naturels positifs
$\mathbb{Q}$ ..... Ensemble des rationnels
$\mathbb{R}$ ..... Ensemble des réels
$\mathbb{R}_+$ ..... Ensemble des réels positifs
$\mathbb{Z}$ ..... Ensemble des entiers relatifs

INTRODUCTION

De nombreux problèmes théoriques ou issus de la pratique peuvent être modélisés comme des problèmes de réalisabilité ayant certaines contraintes bien définies. Nous verrons dans la suite du texte que ces problèmes sont appelés des problèmes de satisfaction de contraintes ou CSP (de l'anglais *constraint satisfaction problems*).

Les CSP ont d'abord été développés dans le cadre de l'intelligence artificielle, mais servent maintenant à modéliser de nombreux problèmes d'optimisation. Des algorithmes et des logiciels ont été développés pour résoudre les CSP en général ou pour résoudre certains CSP en particulier.

Il arrive fréquemment que les contraintes définissant un problème soient non réalisables, en ce sens qu'il n'existe aucune solution satisfaisant l'ensemble des contraintes du problème. Dans ces cas-là, il est intéressant et utile de pouvoir déterminer un sous-ensemble de contraintes ou de variables du problème original tel que ce sous-ensemble soit aussi non réalisable et qu'il soit, si possible, de taille plus petite que le problème de départ. Un tel sous-ensemble peut, par exemple, être utile pour prouver la non réalisabilité du problème de départ ou pour savoir comment modifier le problème initial dans le but de le rendre réalisable (afin d'obtenir la réalisabilité, la recherche d'un tel sous-ensemble devra peut-être être répétée plusieurs fois pour éliminer toutes les causes rendant le problème non réalisable).

Par ailleurs, il arrive que lorsqu'un CSP est réalisable, certaines valeurs du domaine des variables ne participent jamais à une solution vérifiant une contrainte spécifique. Dans de tels cas, il est intéressant de développer un algorithme permettant de supprimer ces valeurs.

Dans cette thèse, nous étudions plusieurs CSP et différentes contraintes spécifiques de CSP. Nous présenterons des algorithmes permettant de rechercher des sous-ensembles

non réalisables irréductibles pour les CSP non réalisables et un algorithme permettant de supprimer les valeurs impossibles des variables pour une contrainte particulière.

Dans la suite de cette introduction, nous présentons de manière informelle le contexte global dans lequel se situe la thèse et les problèmes qui vont être étudiés dans la thèse. Toutes les définitions formelles seront données dans le chapitre 2. À la fin de cette introduction, nous présentons les objectifs et la structure de la thèse.

1.1 Contexte global

Les problèmes que traiterons dans cette thèse peuvent être modélisés comme des problèmes de satisfaction de contraintes ou CSP. Ces problèmes sont définis au moyen de variables, chaque variable ayant un domaine de valeurs admissibles, et de contraintes limitant les combinaisons possibles des variables. La définition formelle des CSP est donnée à la section 2.2.

Nous travaillerons avec les CSP "théoriques" suivants : le problème de satisfaisabilité booléenne communément appelé problème SAT (la définition formelle est donnée à la section 2.3), le problème de k-coloration de graphes (définition formelle donnée à la section 2.1.1) et le problème de D-coloration de graphes (définition formelle donnée à la section 2.1.2). Afin de rendre les définitions de CSP plus concrètes, nous présenterons un CSP définissant un problème réel de confection d'horaires pour le personnel navigant aérien. Ce problème va pouvoir être modélisé en utilisant un CSP où les variables sont les tâches à effectuer et les domaines sont les employés pouvant effectuer les tâches. Les contraintes sont de trois types : il y a des contraintes de non-chevauchement entre des tâches ayant lieu en même temps, des contraintes de qualifications et des contraintes

de charge de travail minimale et maximale pour les employés. Nous travaillerons aussi avec une contrainte globale de CSP : la contrainte SomeDifferent qui impose à certaines variables bien définies d'obtenir des valeurs distinctes.

Quand un problème de satisfaction de contraintes est non réalisable (i.e., quand il n'est pas possible d'attribuer à chaque variable une valeur appartenant à son domaine de telle sorte que toutes les contraintes soient satisfaites), il est très intéressant et très utile de pouvoir extraire un sous-ensemble de contraintes ou de variables du problème original de telle sorte que le sous-problème induit par le sous-ensemble de contraintes ou de variables soit aussi non réalisable et qu'il soit irréductible (i.e., que si on enlève n'importe quelle contrainte ou n'importe quelle variable du sous-problème celui-ci devient réalisable). Nous allons appeler de tels sous-ensembles des sous-ensembles incohérents irréductibles de contraintes ou de variables et nous utiliserons l'abréviation IIS (de l'anglais *infeasible irreducible subset*) de contraintes ou de variables.
Dans cette thèse, nous allons chercher des IIS de contraintes et de variables pour le problème SAT et des IIS de contraintes pour le problème de confection d'horaires pour le personnel navigant aérien. En effet, il est très courant que le CSP modélisant le problème de confection d'horaires pour le personnel navigant aérien soit non réalisable. Afin que la personne gestionnaire de la création de tels horaires soit capable de modifier certaines contraintes ou certains domaines, nous voulons mettre en place des outils de détection automatique de sous-problèmes incohérents minimaux.

Quand un problème de satisfaction de contraintes est réalisable, il arrive fréquemment que certaines valeurs des domaines des variables ne participent jamais à une solution satisfaisant toutes les contraintes du problème. Il est alors très intéressant de pouvoir supprimer ces couples variable-valeur impossibles. Nous verrons à la section 2.4.2 que nous appelons ceci le filtrage des domaines. Dans cette thèse, nous développerons un algorithme de filtrage pour une contrainte globale de CSP : la contrainte SomeDifferent.

Nous verrons au chapitre 2 que la plupart des problèmes que nous traiterons dans cette thèse sont NP-difficiles ou NP-complets. Pour cette raison, nous allons utiliser des méthodes de recherche locale et plus particulièrement l'algorithme de recherche tabou. Nous supposons dans cette thèse que le lecteur est familier avec les méthodes de recherche locale, mais les personnes qui veulent se rafraîchir la mémoire peuvent consulter l'annexe I à la page 260 dans laquelle sont présentées les méthodes de recherche locale et plus particulièrement l'algorithme de descente et l'algorithme de recherche tabou.

1.2 Objectifs de cette thèse

Les principaux objectifs de cette thèse sont les suivants.
Nous développerons trois algorithmes de détection d'IIS de contraintes ou de variables pour le problème SAT et deux algorithmes de détection d'IIS de contraintes pour le problème de confection d'horaires pour le personnel navigant aérien. Nous utiliserons principalement des algorithmes de recherche tabou pour l'implémentation de tels algorithmes. Dans le cas du problème de confection d'horaires, nous présenterons aussi un algorithme exact permettant de vérifier la non réalisabilité des sous-ensembles obtenus. De plus, nous allons aussi mettre en oeuvre des techniques permettant de supprimer les couples variable-valeur impossibles pour la contrainte SomeDifferent. À nouveau, nous allons en partie utiliser l'algorithme de recherche tabou pour effectuer ce filtrage. Pour le problème de confection d'horaires pour le personnel navigant, nous testerons si ces outils de filtrage permettent de gagner du temps lors de la recherche d'IIS lorsque le filtrage est effectué avant la recherche d'IIS.

1.3 Organisation de la thèse

Nous allons maintenant décrire le plan de la thèse.
Dans le deuxième chapitre, sont présentées les définitions formelles des différentes notions et des différents problèmes utilisés dans la thèse : problèmes de coloration de graphes, problèmes de satisfaction de contraintes, problème SAT, programmation par contraintes, filtrage, contraintes AllDifferent et SomeDifferent et sous-ensembles incohérents irréductibles (IIS).

Le troisième chapitre est une revue de la littérature des techniques existantes de recherche d'IIS pour différents problèmes. Premièrement, sont décrites des techniques générales existantes de détection d'IIS pour les problèmes de satisfaction de contraintes. Puis, nous décrivons des méthodes existantes pour l'extraction d'IIS pour le problème de k-coloration. Après cela, nous présentons quelques méthodes de résolution du problème SAT et des méthodes de recherche de sous-ensembles incohérents minimaux pour SAT.

Dans le quatrième chapitre, nous décrivons les méthodes développées par Galinier et Hertz [48] dans le but de trouver des sous-ensembles incohérents minimaux. Nous expliquons comment nous avons adapté ces méthodes pour le problème SAT et nous montrons comment nous utilisons l'algorithme tabou pour résoudre un problème Max-SAT pondéré. Des techniques permettant d'accélérer la recherche d'IIS ou de trouver des IIS plus petits sont aussi données. Finalement, nous donnons de nombreux résultats expérimentaux pour illustrer le comportement de nos méthodes sur différents types d'instances.

Dans le chapitre 5, nous faisons une revue de la littérature de certaines méthodes de filtrage existantes pour les contraintes globales AllDifferent et SomeDifferent et pour quelques autres contraintes globales de CSP.

Dans le sixième chapitre, nous présentons les outils développés pour le filtrage des valeurs impossibles des domaines des variables pour la contrainte SomeDifferent. Nous montrons comment nous avons adapté un algorithme tabou afin de pouvoir filtrer très rapidement le plus de valeurs possible et comment nous avons utilisé un algorithme exact afin de vérifier les valeurs non filtrées par l'algorithme de recherche tabou. Nous présentons également des techniques de réduction de graphes permettant d'accélérer ce filtrage. Finalement, nous présentons des résultats expérimentaux sur des instances provenant de problèmes réels et sur des instances aléatoires.

Dans le chapitre 7, nous détaillons le problème de confection d'horaires pour le personnel navigant dans le transport aérien et présentons une revue de littérature des outils existants pour résoudre ce problème.

Dans le huitième chapitre, nous exposons comment les outils présentés aux chapitres 4 et 6 peuvent être adaptés au problème de confection d'horaires pour le personnel navigant aérien. Pour cela, nous présentons comment nous avons adapté les méthodes de recherche d'IIS pour ce problème particulier. Ensuite, nous montrons comment nous avons adapté un algorithme tabou pour effectuer la recherche d'IIS. Afin de pouvoir vérifier les IIS trouvés, nous montrons comment nous avons modifié un algorithme exact pour la k-coloration de graphes afin de pouvoir tenir compte des contraintes supplémentaires de ce problème réel, ce qui nous permet d'avoir un algorithme exact pour vérifier les supposés IIS fournis par nos heuristiques. Finalement, nous présentons différents résultats expérimentaux.

À la fin de la thèse, nous présenterons une discussion et une conclusion mettant en relief ce qui a été fait et mettant en perspective ce qui pourrait être effectué dans le futur.

CHAPITRE 2

NOTIONS PRÉLIMINAIRES

Dans ce chapitre, nous présentons les définitions formelles des différentes notions nécessaires à la compréhension de la thèse ainsi que les méthodes de programmation par contraintes. Nous commençons d'abord avec les problèmes de coloration de graphes, puis nous présentons les problèmes de satisfaction de contraintes et ensuite le problème SAT de satisfaisabilité booléenne. À la section 2.4, nous présentons la programmation par contraintes. Ensuite, nous présentons les contraintes globales AllDifferent et Some-Different. La dernière section de ce chapitre présente les définitions de sous-ensembles incohérents minimaux de contraintes et de variables.

2.1 Problèmes de coloration de graphes

Dans cette section, nous présentons les problèmes de k-coloration de graphes et de D-coloration de graphes.

2.1.1 Problème de k-coloration de graphes

Dans cette section, nous présentons le problème de k-coloration de graphes.
Soit $G = (V, E)$ un graphe composé d'un ensemble de sommets V (vertices) et d'un ensemble d'arêtes E (edges) entre certaines paires de sommets.

La coloration des sommets d'un graphe consiste à attribuer une couleur à chaque sommet du graphe de telle sorte que deux sommets voisins (i.e., reliés par une arête) reçoivent

des couleurs différentes.

Définition 2.1.1. Une *k-coloration de* G , $k \in \mathbb{N}$, est une fonction $c : V \rightarrow \{1, \ldots, k\}$ qui attribue une couleur $c(v)$ à chaque sommet $v \in V$.

Définition 2.1.2. Étant donné une k-coloration de G, une arête est *en conflit* si ses deux extrémités ont la même couleur. La couleur de ces deux sommets est aussi *en conflit.*

Définition 2.1.3. Une k-coloration de G est *légale* si elle n'a aucun conflit, i.e., $c(i) \neq c(j) \; \forall \; (i,j) \; \in E$.

Définition 2.1.4. Le *nombre chromatique* $\chi(G)$ d'un graphe G, est le plus petit nombre entier k tel qu'il existe une k-coloration légale de G (donc c'est le plus petit nombre de couleurs dont on a besoin pour colorer légalement les sommets de G sans créer de conflit).

Le graphe présenté dans la Figure 2.1 peut être coloré en trois couleurs différentes mais pas en deux couleurs à cause du triangle abf ou du pentagone $bcdef$. Son nombre chromatique est donc 3.

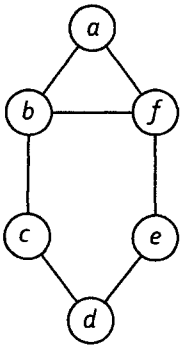


Figure 2.1 – Graphe ayant un nombre chromatique 3.

Le problème consistant à déterminer le nombre chromatique d'un graphe G est connu sous le nom de *problème de coloration des sommets d'un graphe* et est un problème NP-difficile [55].

Définition 2.1.5. *Le problème de k-coloration de graphe* consiste à trouver une k-coloration légale ou à montrer qu'il n'en existe aucune.

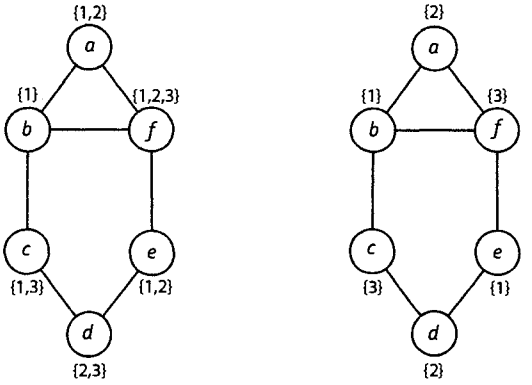
2.1.2 Problème de D-coloration de graphes

Nous considérons maintenant un problème de coloration d'un graphe G où chaque sommet v peut obtenir seulement une couleur appartenant à un domaine D_v et où les domaines de tous les sommets ne sont pas forcément les mêmes.
Plus formellement, soit un graphe $G = (V, E)$ avec l'ensemble de sommets $V = \{1, ..., n\}$ ayant pour domaines $D = \{D_1, ..., D_n\}$, et l'ensemble d'arêtes E. Nous dirons que D_v est l'*ensemble des couleurs* de v et nous utiliserons la notation $D(U) = \bigcup_{v \in U} D_v$ pour chaque $U \subseteq V$.

Définition 2.1.6. *Une D-coloration de G* est une fonction $c : V \rightarrow D(V)$ qui affecte une couleur $c(v) \in D_v$ à chaque sommet de telle sorte que $c(u) \neq c(v)$ pour chaque arête $(u, v) \in E$

Le graphe G est *D-colorable* si une telle affectation existe. Le *problème de coloration par liste* consiste à déterminer si un graphe G, ayant les ensembles de couleurs D, est D-colorable. C'est un problème NP-complet, même s'il est restreint aux graphes d'intervalles [21] ou aux graphes bipartis [79].

Un exemple de problème de D-coloration est présenté dans la Figure 2.2. Les couleurs appartenant au domaine de chaque sommet sont indiquées au-dessus des sommets entre accolades. Ainsi, $D_a = \{1, 2\}$, $D_b = \{1\}$, $D_c = \{1, 3\}$, $D_d = \{2, 3\}$, $D_e = \{1, 2\}$ et $D_f = \{1, 2, 3\}$. Il existe une seule D-coloration légale c, dans laquelle les sommets obtiennent les couleurs suivantes $c(a) = 2$, $c(b) = 1$, $c(c) = 3$, $c(d) = 2$, $c(e) = 1$ et $c(f) = 3$.



(a) Graphe G.

(b) D-coloration de G.

Figure 2.2 – Graphe G ayant une D-coloration (a) et son unique D-coloration légale (b).

2.2 Les problèmes de satisfaction de contraintes

Un problème de satisfaction de contraintes [135] consiste à déterminer si un ensemble de contraintes agissant sur un ensemble de variables discrètes, chacune de ces variables prenant sa valeur dans un domaine, peut être satisfait. Plus formellement, la définition suivante peut être donnée.

Définition 2.2.1. Un *problème de satisfaction de contraintes* (*constraint satisfaction problem, CSP*), $\mathcal{P}(\mathcal{X}, \mathcal{D}, \mathcal{C})$, consiste en :

1. Un ensemble de n variables $\mathcal{X} = \{x_1, ..., x_n\}$.
2. Pour chaque variable $x_i \in \mathcal{X}$ est défini un domaine de valeurs admissibles $D_i = Dom[x_i] \in \mathcal{D}$.
3. Un ensemble de m contraintes $\mathcal{C} = \{C_1, ..., C_m\}$ agissant sur les variables $\mathcal{X}$. Chaque contrainte C_j définit les relations entre les variables en limitant les combinaisons possibles de leurs valeurs.

Le problème consiste à savoir s'il est possible d'attribuer à chaque variable $x_i \in \mathcal{X}$ une valeur d_i appartenant à son domaine D_i, telle que chaque contrainte $C_j \in \mathcal{C}$ soit satisfaite.

Définition 2.2.2. Un état du problème est défini par une *affectation* à chaque variable d'une valeur appartenant à son domaine. Une affectation est aussi appelée dans la littérature une *assignation*, une *instanciation* ou une *interprétation* du problème. Une affectation qui ne viole aucune contrainte est dite *réalisable*, *consistante*, *cohérente* ou *légale*. Une telle affectation est appelée une *solution* de $\mathcal{P}$ ou un *modèle* de $\mathcal{P}$.

Définition 2.2.3. Soit une variable $x_i \in \mathcal{X}$. Une valeur $d \in D_i$ est dite *supportée* s'il existe une solution de $\mathcal{P}$ dans laquelle x_i a la valeur d. Parfois, on dit que d est un support de x_i.

Définition 2.2.4. Un CSP qui a au moins une solution est dit *satisfiable* en anglais. Dans la suite du texte, nous allons utiliser le terme *réalisable*. Les synonymes de réalisable qui se retrouvent dans la littérature sont satisfaisable, cohérent ou consistant.
De même, un CSP qui n'est pas satisfaisable est dit *unsatisfiable* en anglais et nous allons utiliser par analogie le terme *non réalisable* pour le reste de la thèse. Les synonymes de non réalisables utilisés dans la littérature sont insatisfaisable, incohérent ou inconsistant.

Le problème de k-coloration de graphe présenté à la section 2.1.1 est un CSP. En effet, un des encodages possibles consiste à considérer le CSP où chaque variable $x \in \mathcal{X}$ correspond à un sommet $v \in V$, les domaines des variables sont composés des couleurs $\{1, \cdots, k\}$ et il y a une contrainte binaire $C \in \mathcal{C}$ par arête $(i, j) \in E$ du graphe. La contrainte C est satisfaite si $c(i) \neq c(j)$.
De même, le problème de D-coloration est aussi un CSP. La seule différence avec le problème de k-coloration est que les domaines des variables $\mathcal{X}$ ne sont pas tous les mêmes et correspondent aux domaines des sommets V du problème de D-coloration.

De nombreux autres problèmes peuvent s'exprimer comme des problèmes de satisfaction de contraintes, comme par exemple le problème SAT, les problèmes d'ordonnancement ou de confection d'horaires. Le problème de confection d'horaires pour le personnel navigant aérien que nous présenterons au chapitre 7 est un CSP qui peut se modéliser au moyen d'un problème de D-coloration avec contraintes additionnelles (de qualifications et de crédits de vol minimum et maximum).

Définition 2.2.5. *Une contrainte globale d'un CSP* est une contrainte complexe qui agit sur plusieurs ou toutes les contraintes du CSP.

Il existe de nombreuses contraintes globales comme par exemple les contraintes All-Different et SomeDifferent. Ces contraintes sont présentées de façon détaillée à la section 2.4.5.

Nous allons maintenant définir de façon plus détaillée un autre problème pouvant s'exprimer comme un CSP : le problème SAT.

2.3 Problème SAT de satisfaisabilité booléenne

Avant de pouvoir donner la définition formelle du problème SAT, nous allons introduire certaines notions.

Soit un ensemble de n variables booléennes $\mathcal{X} = \{x_1, x_2, ..., x_n\}$. Cela veut dire que le domaine de chaque variable x_i est $D_i = \{vrai, faux\} = \{1, 0\}$ pour tout $i = 1, \ldots n$.
Un *littéral* l_i est une variable x_i (littéral positif) ou sa négation $\bar{x}_i$ (littéral négatif).
Une *clause* C est une disjonction de littéraux, comme par exemple

$$C = x_1 \vee x_2 \vee \bar{x}_3 \vee x_4.$$

La longueur d'une clause C est le nombre de littéraux qui la composent et est notée $|C|$. Si $|C| = k$, alors C est une *k-clause*.

Définition 2.3.1. Une clause est *unitaire* si sa longueur est un, i.e., elle est composée d'un seul littéral.

Définition 2.3.2. *Une formule propositionnelle $\mathcal{F}$ est dans la forme normale conjonctive (Conjunctive Normal Form,* ou *CNF)* si c'est une conjonction de m clauses d'un ensemble $\mathcal{C} = \{C_1, ..., C_m\}$,

$$\mathcal{F} = C_1 \wedge C_2 \wedge ... \wedge C_m.$$

$$\mathcal{F} = \bigwedge_{i=1}^{m} (\bigvee_{j_i=1}^{|C_i|} l_{j_i}).$$

Le problème est défini si on connaît ses m clauses.

Définition 2.3.3. On dit qu'une clause est *tautologique* si elle contient un littéral l et sa négation $\bar{l}$.

La clause vide, $C = \emptyset$, notée aussi $\square$, est interprétée comme fausse. Par contre, la formule conjonctive vide, $\mathcal{F} = \emptyset$, est interprétée comme vraie.
La longueur d'une formule $\mathcal{F}$, est la somme des longueurs de ses clauses, notée $|\mathcal{F}| = \sum_{i=1}^{m} |C_i|$.

Dans ce travail, nous faisons l'hypothèse que toutes les formules booléennes utilisées sont sous la forme normale conjonctive (CNF). Ainsi, nous pouvons utiliser l'appellation SAT, sans préciser SAT-CNF.

Définition 2.3.4. Étant donné n variables et m clauses, le *problème SAT* consiste à trouver une affectation des variables de $\mathcal{X}$ de telle sorte que la formule $\mathcal{F}$ soit évaluée à *vraie*

ou à montrer qu'aucune affectation de la sorte ne peut être trouvée. Le problème est dit *réalisable* si une affectation évaluée à vraie existe, sinon il est dit *non réalisable.*

Le problème SAT est bien un CSP. Il existe plusieurs façons de l'encoder comme un CSP [137]. L'encodage le plus proche de la formulation CNF donne un CSP qui a les variables $\mathcal{X}$ avec les domaines $\{vrai, faux\}$. Les contraintes sont les clauses.

Exemple 2.3.5.

Soit la formule CNF $\mathcal{F}_1$.

$$\begin{aligned} \mathcal{X} &= \{x_1, x_2, x_3\} \\ \mathcal{C} &= \{C_1, C_2, C_3\} \\ \mathcal{F}_1 &= C_1 \wedge C_2 \wedge C_3 \\ \text{où} & \\ C_1 &= x_1 \vee x_2 \\ C_2 &= x_2 \vee \bar{x}_3 \\ C_3 &= \bar{x}_2 \vee x_3 \end{aligned}$$

La formule $\mathcal{F}_1$ de l'Exemple 2.3.5 est réalisable. En effet, il suffit d'affecter les valeurs suivantes aux variables $x_1 = 1$, $x_2 = 1$ et $x_3 = 1$ pour que les trois contraintes C_1, C_2 et C_3 soient satisfaites.

Par contre, la formule $\mathcal{F}_2$ de l'Exemple 2.3.6 est non réalisable. En effet, si nous posons $x_1 = 1$, alors les clauses C_1 et C_3 sont satisfaites. Pour satisfaire C_2 il faut imposer $x_2 = 0$ et pour satisfaire C_4 il faut imposer $x_3 = 0$. Ceci implique que C_5 n'est pas satisfaite. Si à l'origine, nous posons $x_1 = 0$, alors les clauses C_2 et C_4 sont satisfaites. Pour satisfaire C_1 il faut imposer $x_2 = 1$ et pour satisfaire C_3 il faut imposer $x_3 = 1$. Ceci implique que C_6 n'est pas satisfaite.

Exemple 2.3.6.

Soit la formule CNF $\mathcal{F}_2$.

$$
\begin{aligned}
\mathcal{X} &= \{x_1, x_2, x_3\} \\
\mathcal{C} &= \{C_1, C_2, C_3, C_4, C_5, C_6\} \\
\mathcal{F}_2 &= C_1 \wedge C_2 \wedge C_3 \\
&\text{où} \\
C_1 &= x_1 \vee x_2 \\
C_2 &= \bar{x}_1 \vee \bar{x}_2 \\
C_3 &= x_1 \vee x_3 \\
C_4 &= \bar{x}_1 \vee \bar{x}_3 \\
C_5 &= x_2 \vee x_3 \\
C_6 &= \bar{x}_2 \vee \bar{x}_3
\end{aligned}
$$

SAT fait partie du premier groupe de problèmes NP-complets connus (démontré par Cook en 1971 [32]). Selon Gu et al. [70], le problème SAT est très important par exemple dans la logique, l'informatique théorique, la vision par ordinateur ou les bases de données. De plus, le problème SAT est utilisé dans les domaines suivants : la vérification et le diagnostic d'erreurs dans le matériel informatique [85, 89, 90, 126, 128, 129], l'existence de quasi-groupes (les quasi-groupes modélisent une grande variété de problèmes réels tels que la confection d'horaires de tournois) [138, 139], la planification [86], l'ordonnancement [34] ainsi que dans les circuits intégrés [45]. Des problèmes comme par exemple la k-coloration de graphes peuvent être transformés en problèmes SAT afin d'être résolus [57].

Des restrictions peuvent être imposées sur la longueur des clauses. Une formule normale conjonctive où chaque clause est composée exactement de k littéraux est appelée un *k-SAT* ou *k-CNF*. Le problème *3-SAT* est donc un problème *SAT* où chaque clause contient exactement 3 littéraux. Le problème *k-SAT* est NP-complet pour $k \geq 3$ mais

est résoluble en temps polynomial pour $k = 2$ [55]. Un algorithme de résolution est dit *polynomial* s'il est garanti de terminer avec un nombre d'étapes qui est une fonction polynomiale de la taille du problème [55].
Le *k-SAT* le plus étudié est le *3-SAT*. On peut, en effet, réduire n'importe quelle formule normale conjonctive en un *3-SAT*.
À part le problème 2-SAT, d'autres classes de problèmes SAT sont résolubles en temps polynomial comme par exemple les instances de Horn (chaque clause ne contient pas plus d'un littéral positif) ou instances renommables de Horn (tous les littéraux peuvent être renommés de façon uniforme de telle sorte que l'instance avec les littéraux renommés soit de type Horn).

Définition 2.3.7. Le problème *Max-SAT* consiste à satisfaire le plus grand nombre de contraintes.

Max-SAT est un problème NP-difficile [55] (et contrairement à 2-SAT qui est polynomial, même Max-2-SAT est NP-difficile [131]) et est fondamental pour résoudre plusieurs problèmes pratiques en informatique [72]. C'est une généralisation du problème SAT.

Définition 2.3.8. Le problème *Max-SAT pondéré* (*weighted Max-SAT)* est une variante du problème de satisfaisabilité booléenne où chaque clause $C_i \in C$ a un poids $w_i \in \mathbb{R}$, et le but est de trouver une affectation qui maximise la somme des poids de toutes les clauses satisfaites : $\max \sum_{\substack{i \in \{1,\ldots,m\} \\ C_i \text{ est satisfaite}}} w_i$.

Grâce à ces poids, on peut indiquer une préférence sur les clauses qu'on désire satisfaire. *Max-SAT pondéré* est une généralisation de *Max-SAT* : il suffit en effet de fixer tous les poids égaux à 1 ($w_i = 1, \forall\, i = 1, \ldots, m$) et ainsi on cherche à satisfaire le plus de contraintes possibles. Donc le problème Max-SAT pondéré est aussi un problème

NP-difficile.

Max-SAT a de nombreuses applications en intelligence artificielle, optimisation combinatoire, systèmes experts et pour la cohérence des bases de données [10, 16, 17, 30, 51, 72, 108].

Les instances de *k-SAT aléatoires* sont des instances particulières du problème SAT. Trois paramètres sont importants pour les instances de *k-SAT aléatoires* :

1. k = la longueur de chaque clause ;
2. n = le nombre de variables ;
3. m = le nombre de clauses.

Les m clauses sont construites de façon aléatoire. Pour chacune d'entre elles, k littéraux différents sont tirés au hasard. Certains algorithmes se spécialisent uniquement dans la résolution de telles instances.

Un phénomène important se produit avec les *k-SAT aléatoires* : c'est le phénomène du seuil de transition. Si un problème a beaucoup de variables et très peu de clauses, il sera très probablement réalisable et au contraire un problème avec beaucoup de clauses et très peu de variables sera très probablement non réalisable. En fait, la probabilité qu'un problème aléatoire soit réalisable dépend du ratio $\frac{m}{n}$. Il existe un point au-dessous duquel presque toutes les formules sont réalisables et au-dessus duquel elles sont presque toutes non réalisables. Pour les *3-SAT aléatoires*, le seuil a été mesuré de façon expérimentale et se situe à environ 4.24. Ce phénomène est important pour construire des instances difficiles.

2.4 Programmation par contraintes

Dans cette section, nous allons décrire brièvement les définitions de cohérence, le filtrage des domaines, les méthodes de propagation de contraintes, les schémas généraux de résolution des CSP et les contraintes globales AllDifferent et SomeDifferent. Les définitions et les méthodes données dans cette section sont principalement tirées de Bartàk [13–15] et Apt [8].

2.4.1 Techniques de cohérence

Nous décrivons ici des techniques de cohérence qui sont utilisées par les méthodes de programmation par contraintes car ces algorithmes ont généralement pour but d'obtenir une "cohérence locale".

Il existe plusieurs techniques de cohérence [8, 14, 15]. Les noms de ces techniques de cohérence sont dérivés des notions de la théorie des graphes. En effet, les CSP peuvent être représentés au moyen d'un *graphe des contraintes* (de l'anglais *constraint graph*) où les noeuds représentent les variables et les arêtes les contraintes entre deux variables. Afin de pouvoir effectuer cette représentation sous forme de graphe, il faut que le CSP soit sous la forme de *CSP binaire*, i.e., qu'il ne contienne que des contraintes unaires (qui ne concernent qu'une seule variable) ou binaires (qui concernent deux variables). Historiquement, les premières techniques de cohérence ont été développées pour les contraintes binaires principalement pour deux raisons : premièrement, la modélisation de problèmes en utilisant des contraintes globales est apparue après la modélisation de problèmes en utilisant des contraintes binaires, et deuxièmement, parce que chaque CSP peut être transformé en un CSP binaire [13]. C'est pour cette raison que, dans le texte qui suit, nous allons d'abord présenter les principales techniques pour les contraintes unaires ou binaires : cohérence de noeuds, d'arc, directionnelle d'arcs et de chemins. Puis, nous

allons aussi nous intéresser à la cohérence de contraintes globales aussi appelée cohérence de domaine.

Commençons avec la **cohérence de noeuds**.

Définition 2.4.1. Une contrainte unaire C agissant sur la variable x est *noeud-cohérente* si toutes les valeurs du domaine de x sont cohérentes avec C (i.e., sont des solutions de C). Un CSP est *noeud-cohérent* si toutes ses contraintes unaires sont noeud-cohérentes.

Donc la technique permettant d'obtenir la cohérence de noeuds d'un CSP réduit les domaines de chaque variable aux valeurs qui satisfont les contraintes unaires sur ces variables. Quand un CSP est noeud-cohérent, alors les contraintes unaires peuvent être supprimées car les domaines de chaque variable en tiennent compte.

Exemple 2.4.2. *Par exemple, la contrainte* $x > 4$ *où* $D_x = \{3, 4, 5, 6\}$ *n'est pas cohérente par rapport aux noeuds. Il faut enlever les valeurs 3 et 4 du domaine de* x *pour que ce soit cohérent par rapport aux noeuds.*

Étudions maintenant la **cohérence d'arcs** qui concerne les contraintes binaires.

Définition 2.4.3. Soit C une contrainte binaire sur les variables x et y ayant les domaines D_x et D_y. C est dite *arc-cohérente* si

- $\forall\, v_x \in D_x$ alors $\exists\, v_y \in D_y$ tel que C est satisfaite (i.e., $(x = v_x, y = v_y)$ est une solution de C),
- $\forall\, v_y \in D_y$ alors $\exists\, v_x \in D_x$ tel que C est satisfaite (i.e., $(x = v_x, y = v_y)$ est une solution de C).

Définition 2.4.4. Un CSP $\mathcal{P}$ est dit *arc-cohérent* si toutes ses contraintes binaires sont arc-cohérentes.

Exemple 2.4.5. *L'exemple représenté dans la Figure 2.3 sur les variables x et y où $D_x = \{1, 2, 3, 4, 5, 6, 7\}$ et $D_y = \{3, 4, 5, 6\}$ et où la contrainte binaire considérée est $C : x > y$ n'est pas arc-cohérent. En effet, les valeurs 1, 2 et 3 de D_x ne sont pas cohérentes avec C (car elles ne peuvent pas vérifier la contrainte $x > y$).*

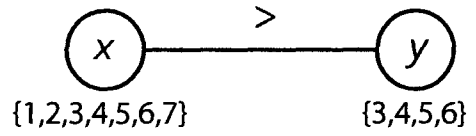


Figure 2.3 – Exemple d'une contrainte binaire non arc-cohérente.

Par contre, l'exemple présenté dans la Figure 2.4 est arc-cohérente. Toutes les valeurs des domaines de D_x et D_y participent à une solution de la contrainte $x > y$.

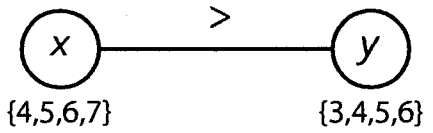


Figure 2.4 – Exemple d'une contrainte binaire arc-cohérente.

Remarquons que les contraintes binaires peuvent ne pas être arc-cohérentes mais le CSP peut être réalisable. Par exemple, Apt [8] propose l'exemple présenté dans la Figure 2.5. En effet, afin d'obtenir l'arc-cohérence pour cet exemple, il faut enlever la valeur b du domaine de x.

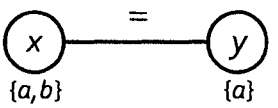


Figure 2.5 – Exemple d'un CSP réalisable avec une contrainte binaire non arc-cohérente.

De même, le cas contraire peut aussi se produire, i.e., qu'un CSP, où toutes les contraires binaires sont arc-cohérentes, peut être non réalisable . Apt [8] propose l'exem-

ple présenté dans la Figure 2.6 pour illustrer ceci. Les deux contraintes sont en effet arc-cohérentes car toutes les valeurs des domaines de x et y peuvent participer à une solution concernant chaque contrainte individuellement, par contre le CSP est non réalisable.

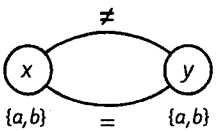


Figure 2.6 – Exemple d'un CSP non réalisable où les contraintes binaires sont arc-cohérentes.

Les techniques permettant d'obtenir l'arc-cohérence sur une contrainte binaire C agissant sur les variables x et y enlèvent les valeurs des domaines des variables x et y qui sont incohérentes avec la contrainte C.

Cette notion d'arc-cohérence peut être modifiée pour tenir compte d'un ordre linéaire $\prec$ (en anglais *linear ordering*) sur les variables considérées. L'idée est que l'existence de supports est requise dans un sens seulement. Cela permet de donner la définition d'arc-cohérence directionnelle.

Définition 2.4.6. Soit C une contrainte binaire sur les variables x et y ayant les domaines D_x et D_y. C est dite *directionnellement arc-cohérente avec le respect de* $\prec$ si exactement une des deux conditions suivantes est vérifiée pour C

- **si** $x \prec y$ **:** $\forall\, v_x \in D_x$ alors $\exists\, v_y \in D_y$ tel que C est satisfaite étant donné $x \prec y$ (i.e., $(x = v_x, y = v_y)$ est une solution de C),
- **si** $y \prec x$ **:** $\forall\, v_y \in D_y$ alors $\exists\, v_x \in D_x$ tel que C est satisfaite étant donné $y \prec x$ (i.e., $(x = v_x, y = v_y)$ est une solution de C).

Un CSP est dit directionnellement arc-cohérent avec le respect de $\prec$ si toutes ses contraintes binaires sont directionnellement arc-cohérentes avec le respect de $\prec$.

Apt donne l'exemple suivant d'une contrainte directionnellement arc-cohérente.

Exemple 2.4.7. *Considérons la contrainte représentée dans la Figure 2.7. Cette contrainte est directionnellement arc-cohérente étant donné* $y \prec x$ *(par contre elle n'est pas arc-cohérente). En effet, pour chaque valeur* $v_y \in \{3, 4, 5, 6, 7\}$ *il existe une valeur* $v_x \in \{2, 3, 4, 5, 6, 7, 8, 9, 10\}$ *telle que* $x < y$.

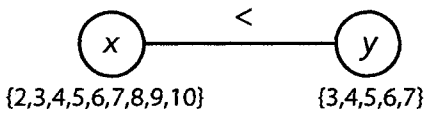


Figure 2.7 – Exemple d'une contrainte directionnellement arc-cohérente étant donné $y \prec x$.

D'autres valeurs peuvent être enlevées des domaines des variables grâce aux techniques concernant la cohérence de chemin. Afin de pouvoir donner cette définition, introduisons les notations suivantes. Soit $C_{x,y}$ l'ensemble des contraintes binaires sur les variables x et y.

Définition 2.4.8. Un CSP est dit *chemin-cohérent* si pour chaque sous-ensemble $\{x, y, z\}$ de ses variables la condition suivante est vérifiée : pour tout $a \in D_x$ et $c \in D_z$, si $(x = a, z = c)$ vérifie l'ensemble des contraintes binaires $C_{x,z}$ alors il existe $b \in D_y$ tel que $(x = a, y = b)$ vérifie l'ensemble des contraintes binaires $C_{x,y}$ et $(y = b, z = c)$ vérifie l'ensemble des contraintes $C_{y,z}$.

Exemple 2.4.9. *Considérons le CSP représenté dans la Figure 2.8. Ce CSP est chemin-cohérent.*

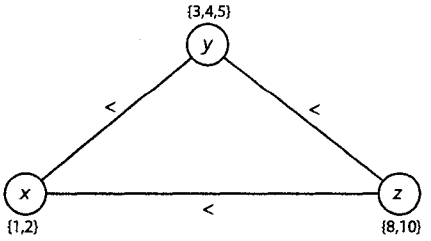


Figure 2.8 – Exemple d'un CSP chemin-cohérent.

Donc la cohérence de chemin requiert que pour chaque paire de valeurs des domaines des variables x et z respectant les contraintes binaires sur x et z, il existe une valeur pour chaque variable y le long d'un chemin entre x et z telle que toutes les contraintes binaires du chemin soient satisfaites.

Nous pouvons maintenant généraliser la notion de cohérence d'arcs pour des contraintes globales : c'est la notion de cohérence de domaine. Intuitivement une contrainte C est domaine-cohérente si toutes les valeurs des domaines des variables sur lesquelles C agit participent à une solution de C. Plus formellement, nous avons la définition suivante.

Définition 2.4.10. Soit une contrainte C agissant sur les variables $x_1, \ldots, x_n$ ayant les domaines respectifs $D_1, \ldots, D_n$. C est dite *domaine-cohérente* si pour chaque $i \in [1, \ldots, n]$ et $a \in D_i$ il existe $l_1 \in D_1, \ldots, l_{i-1} \in D_{i-1}, l_{i+1} \in D_{i+1}, \ldots, l_n \in D_n$ tels que la contrainte C soit satisfaite.
Un CSP est dit *domaine-cohérent* si toutes ses contraintes globales sont domaine-cohérentes.

Cette notion de domaine-cohérence est appelée parfois hyper-arc cohérence ou arc-cohérence généralisée et est équivalente à l'arc-cohérence mais pour les contraintes globales.
En s'assurant de la cohérence de domaine d'une contrainte globale, le niveau de cohérence locale obtenu est plus fort que celui obtenu en s'assurance de la cohérence d'arc des contraintes binaires équivalentes à la contrainte globale. Pour montrer ceci, considérons l'exemple suivant : les variables sont $\{x_1, x_2, x_3\}$ et les domaines des variables sont : $D_1 = \{1, 2\}, D_2 = \{1, 2\}, D_3 = \{1, 2\}$. Si nous considérons les contraintes binaires suivantes $x_1 \neq x_2$, $x_1 \neq x_3$, $x_2 \neq x_2$, alors ces trois contraintes sont arc-cohérentes, par contre il n'existe pas de solution. La contrainte globale AllDifferent(x_1, x_2, x_3) modélisant exactement le même CSP n'est pas domaine-cohérente.

Dans cette thèse, la notion de cohérence locale que nous cherchons à obtenir est la cohérence de domaine puisque les contraintes que nous allons utiliser sont des contraintes globales.

2.4.2 Filtrage des domaines

Les méthodes consistant à obtenir la cohérence de domaine pour une contrainte globale de CSP sont généralement appelées *méthodes de filtrage* ou *méthodes de filtrage des domaines*. Ces méthodes consistent à enlever certaines valeurs des domaines des variables qui ne peuvent jamais être attribuées aux variables dans une solution légale. Par exemple, dans la Figure 2.9, le sommet c ne peut jamais obtenir la couleur 1 ou la couleur 2, car les deux sommets qui lui sont adjacents a et b sont obligatoirement colorés avec ces couleurs (la seule couleur possible pour a est 2 et la seule couleur possible pour b est 1). Dans ces cas-là, ces valeurs peuvent être supprimées des domaines des variables. Dans le cas de la Figure 2.9, le domaine de c peut donc être réduit à $D_c = \{3\}$.

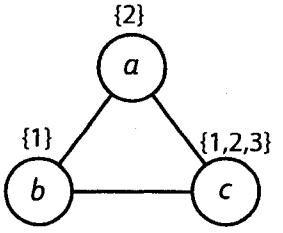


Figure 2.9 – Problème de D-coloration où certaines valeurs du domaine du sommet c peuvent être filtrées.

Dans ce travail, nous allons présenter des outils pour faire du filtrage de domaine pour la contrainte globale SomeDifferent présentée à la section 2.4.5. Dans notre cas, ceci va être effectué comme un prétraitement général pour accélérer les méthodes de détection d'IIS et non comme méthode de résolution du CSP.

2.4.3 Propagation de contraintes

Les algorithmes qui permettent d'obtenir la cohérence locale sont appelés des algorithmes de propagation de contraintes. Comme Apt le précise dans [8], dans la littérature de nombreux autres noms ont été donnés pour ces algorithmes, en voici quelques exemples (avec à chaque fois le terme en anglais entre parenthèses) : algorithmes de cohérence (consistency algorithms), algorithmes de cohérence locale (local consistency algorithms), algorithmes de propagation locale (local propagation algorithms), algorithmes d'imposition de cohérence (consistency enforcing algorithms), algorithmes de filtrage (filtering algorithms), algorithme de filtrage des domaines et algorithmes de rétrécissement (narrowing algorithms). Comme nous le verrons à la section 2.4.4, ces algorithmes forment un élément principal du schéma général des algorithmes de programmation par contraintes.

La propagation de contrainte consiste à supprimer des valeurs des domaines des variables ne participant à aucune solution du problème. Un algorithme de propagation de contraintes traite les différentes contraintes du problème l'une après l'autre, il identifie et supprime les valeurs des domaines des variables qui n'apparaissent dans aucune solution de la contrainte considérée selon le niveau de cohérence locale recherché. Ainsi, l'espace de recherche devient plus petit. Il se peut que certaines des variables qui ont vu leur domaine réduit par l'algorithme apparaissent dans d'autres contraintes. Alors, ces contraintes sont traitées par l'algorithme de propagation et le même processus est effectué sur elles. L'effet de réduction des domaines est donc propagé. Quand, à une certaine étape de cette recherche, aucune nouvelle valeur incohérente ne peut être filtrée, l'algorithme a alors atteint un certain niveau de cohérence locale. Comme nous l'avons vu, il y a plusieurs niveaux de cohérence locale qui peuvent être atteints, chacun supprimant plus ou moins de valeurs des domaines. Afin d'avoir un algorithme de propagation de contraintes efficace, il faut trouver le bon compromis entre le niveau de cohérence locale

obtenu et le temps pour obtenir cette cohérence locale.
Parfois un CSP peut être résolu uniquement en utilisant un algorithme de propagation de contraintes. En effet, il est possible qu'après avoir obtenu un certain niveau de cohérence locale des contraintes, il ne reste qu'une seule valeur dans le domaine de chaque contrainte et que ces valeurs forment une solution du CSP.

Selon Bartàk [14], il y a deux familles principales de schémas de vérification de cohérence : le schéma *regard en arrière* (en anglais *look back*) et le schéma par *anticipation* (en anglais *look ahead*).
Le schéma "regard en arrière" utilise des vérifications de cohérence parmi les variables déjà affectées. Un simple schéma de retour-arrière, abrégé fréquemment BT pour backtracking, utilise cette stratégie dans le sens que si une affectation est faite sur une variable, il s'assure que l'affectation faite est compatible avec les valeurs déjà données aux variables jusque-là et si ce n'est pas possible effectue un retour-arrière. D'autres techniques utilisant un schéma de "regard en arrière" sont la méthode de *saut en arrière* (abrégée BJ en fonction du terme anglais *backjumping*) ou *marquage arrière* (abrégé BM pour le terme anglais *backmarking*).
Le principal désavantage de ces méthodes est la détection tardive des incohérences, i.e., elles résolvent l'incohérence quand elle se produit mais ne la préviennent pas. C'est pour cela que les méthodes appelées *d'anticipation* (de l'anglais *look ahead*) ont été décrites. Ces techniques préviennent les incohérences futures.

La première de ces techniques est la *vérification vers l'avant* (abrégé FC de l'anglais *forward checking*). Étant donné une affectation partielle des variables, cette méthode s'assure de la cohérence entre toutes les paires de variables où une des variables est affectée et l'autre n'est pas affectée. Quand une valeur est affectée à une variable, toutes les valeurs des variables non encore affectées et qui sont incohérentes avec cette valeur sont supprimées temporairement du domaine. Donc cette méthode s'assure que, locale-

ment, pour chaque variable non affectée, il y ait au moins une valeur de son domaine qui soit cohérente avec les valeurs des variables déjà affectées.
La stratégie de recherche qui applique l'arc-cohérence à chaque noeud de l'arbre de recherche (i.e., lors de chaque affectation d'une variable) s'appelle *Maintien de la cohérence d'arcs* (abrégé MAC pour le terme anglais *maintaining arc consistency*). La différence entre les stratégies FC et MAC est que la première vérifie la cohérence par rapport à une seule variable non affectée à la fois, tandis que MAC vérifie aussi la cohérence entre les paires de variables non affectées.

Les méthodes de filtrage pour les contraintes globales (telles que les contraintes `All-Different` et `SomeDifferent`) sont généralement plus complexes et très spécifiques aux contraintes. Quelques exemples de telles méthodes sont présentés au chapitre 5 à la page 140.

2.4.4 Recherche d'une solution

Les problèmes de satisfaction de contraintes peuvent être définis sur des domaines finis (comme par exemple le problème de k-coloration de graphes que nous avons vu dans l'introduction) ou sur des domaines infinis. Le terme *programmation par contraintes, abrégé PC ou CP en anglais,* est utilisé pour définir certaines méthodes de résolution des CSP. Bartàk [14] précise qu'il y a deux branches de programmation par contraintes : la première concerne les problèmes pour lesquels les domaines des variables sont finis et la deuxième concerne les problèmes pour lesquels les domaines des variables sont infinis. Dans le cas fini, il utilise le terme *satisfaction de contraintes* (de l'anglais *constraint satisfaction*) et dans le cas des domaines infinis ou plus complexes, il utilise le terme *résolution de contraintes* (de l'anglais *constraint solving*). Dans cette thèse, nous allons considérer uniquement des CSP avec des domaines finis et nous allons simplement utiliser le terme général "programmation par contraintes".

Dans la plupart des problèmes réels, nous ne voulons pas uniquement trouver une solution, mais trouver une bonne solution, nous allons appeler de tels problèmes, *des problèmes d'optimisation sous contraintes*. Cela veut dire que la valeur de la solution est mesurée par une fonction dite *fonction objectif* qui est minimisée ou maximisée selon le problème.

Apt [8] décrit la programmation par contraintes comme un processus de transformation des CSP. En effet, durant la résolution, les CSP sont transformés jusqu'à ce qu'une solution (ou toutes les solutions selon le but recherché) soit trouvée ou jusqu'à ce qu'il soit prouvé qu'il n'existe aucune solution. Les CSP sont transformés de telle sorte à ce qu'ils soient équivalents au CSP original.

La façon la plus simple et la plus naïve de résoudre un CSP est la procédure *générer et tester* (abrégée GT pour le terme en anglais *generate and test*). Cette procédure génère une affectation complète des variables et teste si cette affectation satisfait toutes les contraintes du CSP. Si oui, alors une solution est trouvée et la procédure s'arrête, sinon une autre affectation est générée. Ce processus est répété jusqu'à ce qu'aucune nouvelle affectation ne puisse être générée ou jusqu'à ce qu'une solution soit trouvée. Cette méthode de résolution n'est pas très efficace et elle est généralement utilisée quand toutes les autres méthodes de résolution ont échoué. Afin d'améliorer son efficacité, il est possible d'utiliser des générateurs intelligents (ou informés) par exemple en utilisant des méthodes de recherche locale ou alors de la combiner avec un testeur qui vérifie la réalisabilité d'une contrainte dès que toutes ses variables sont affectées. Cette méthode est utilisée par l'approche avec retour-arrière.

La recherche avec retour-arrière (de l'anglais *backtracking*) part de la racine de l'arbre et le parcourt en profondeur d'abord. Quand une feuille est rencontrée la recherche retourne

en arrière vers le noeud parent de cette feuille et ensuite visite le prochain descendant de ce parent. Ce processus continue jusqu'à ce que la recherche atteigne à nouveau le noeud racine et que tous ses descendants aient été visités. Cette méthode augmente donc de façon incrémentale une solution partielle (dont les valeurs des variables affectées sont cohérentes) vers une solution complète. Les variables sont affectées séquentiellement et dès que toutes les variables impliquées dans une contrainte sont affectées alors la validité de la contrainte est vérifiée. Dès qu'une affectation partielle viole au moins une contrainte, alors un retour-arrière est effectué, et les noeuds en-dessous du noeud courant ne sont pas évalués. La Figure 2.10 montre une recherche avec retour-arrière. Dans

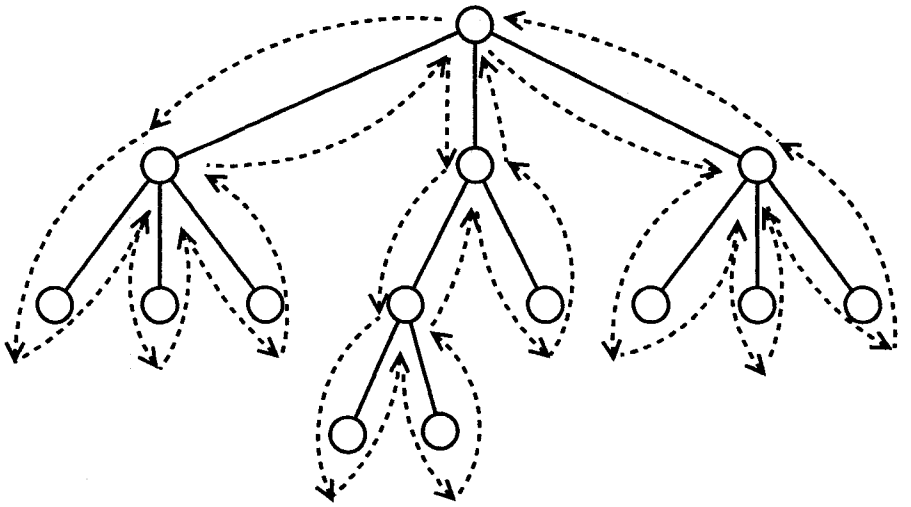


Figure 2.10 – Exemple d'un arbre de recherche avec retour-arrière.

ce cas-ci tous les noeuds et les feuilles de l'arbre de recherche sont des CSP. Dans les feuilles se trouve soit la résolution du CSP (si le CSP est réalisable) soit une preuve de l'incohérence de la solution partielle menant à la feuille. Si le but est de trouver une solution, alors la recherche s'arrête dès qu'une feuille "résolue" est trouvée. Sinon, si le but est de trouver toutes les solutions, alors la recherche continue jusqu'à ce que toutes les feuilles soient visitées.

La recherche par *séparation et évaluation progressive* (de l'anglais *branch-and-bound*)

est une variante du backtracking dans le cas des problèmes d'optimisation sous contraintes. Cela veut dire qu'il y a une fonction objectif et le but est de trouver une solution qui minimise (ou maximise selon les cas) la valeur de la fonction objectif. L'algorithme de séparation et évaluation progressive utilise une fonction heuristique qui évalue l'affectation partielle du noeud courant. Cette fonction heuristique représente une sous-estimation (dans le cas d'une minimisation) de la fonction objectif pour l'affectation partielle. L'algorithme parcourt l'arbre de recherche. Il y a plusieurs façons de parcourir l'arbre de recherche : la façon la plus courante est la recherche en profondeur (*depth-first search*), d'autres parcours sont la recherche en largeur (*breadth-first search*) ou la recherche meilleur d'abord (*best-first search*). Dès que le domaine d'une variable est réduit à une seule valeur (ce qui correspond virtuellement à affecter cette valeur à la variable), la valeur de la fonction heuristique pour cette nouvelle affectation partielle est calculée. Si cette valeur excède la borne actuelle, alors les noeuds enfants du noeud actuel ne sont pas évalués et le sous-arbre est élagué. Au départ, la borne est fixée à $+\infty$ (dans le cas d'une minimisation) et durant la recherche cette valeur est mise à jour et stocke la plus petite valeur rencontrée. La Figure 2.11 montre une recherche de type "séparation et évaluation progressive" où les branches non explorées sont en pointillés.

Nous allons maintenant voir comment sont développés les algorithmes plus évolués de programmation par contraintes.
Apt [8] propose comme schéma général de la programmation par contraintes le schéma présenté dans la Figure 2.12.

La procédure générale est donc récursive. Nous allons maintenant décrire les procédures et les variables qu'elle emploie. La variable *Continuer* est booléenne et permet d'arrêter `Résolution`($\mathcal{P}$) quand la procédure `TestDomaines` retourne vrai. Cette

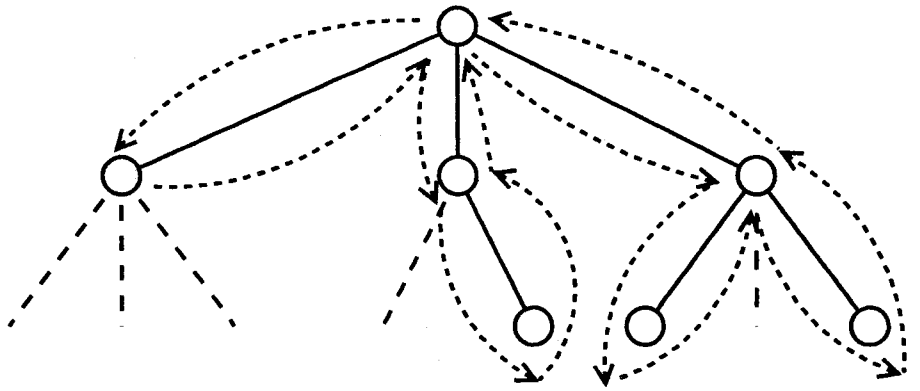


Figure 2.11 – Exemple d'un arbre de séparation et évaluation progressive.

`Résolution`($\mathcal{P}$)

Entrée : Un CSP $\mathcal{P}$.
Sortie : Une solution ou toutes les solutions du CSP ou une preuve qu'il n'en existe aucune.

Continuer ← vrai;
tant que *Continuer est vrai et* `ButAtteint` *est faux* **faire**
  `Prétraitement`;
  `PropagationContraintes`;
  si *non* `ButAtteint` **alors**
    si `TestDomaines` **alors** *Continuer* ← faux;
    sinon
      $\mathcal{P}'$ ← `Partager`($\mathcal{P}$);
      pour chaque $\mathcal{P}'_i \in \mathcal{P}'$ **faire**
        `Résolution`($\mathcal{P}'_i$);

Figure 2.12 – Schéma général de la programmation par contraintes

procédure `TestDomaines` vérifie s'il vaut la peine de partager le CSP. Elle retourne vrai si tous les domaines des variables sont des singletons (auquel cas le CSP est satisfait) ou si un des domaines est vide (auquel cas le CSP n'est pas satisfait). Il est possible d'étendre ce test des domaines et la procédure peut retourner vrai si aucune recherche plus avancée n'est nécessaire (par exemple parce que le CSP est résolu).

La procédure `ButAtteint` retourne vrai si le but initial pour le CSP original est atteint et retourne faux sinon. Le but recherché dépend des applications et est fixé d'avance. Il peut être par exemple : une solution a été trouvée, toutes les solutions ont été trouvées ou une incohérence a été détectée.

La procédure `Prétraitement` a pour but d'amener le CSP original dans la forme syntaxique voulue pour la résolution.

La procédure `Partager` consiste à subdiviser le CSP en deux (ou plus) nouveaux CSP. Ceci peut être fait soit en partageant un domaine ou en partageant une contrainte. Par exemple, pour un problème de k-coloration où la variable x a le domaine $D = \{1, 2, 3, 4\}$. Une première façon de partager le domaine de x pourrait être de le partager de façon à créer deux CSP. Le premier CSP peut avoir comme domaine pour x : $D = \{1\}$ (i.e., la valeur 1 est affectée à x) et le deuxième CSP a comme domaine de x : $D = \{2, 3, 4\}$. Une autre façon de partager ce domaine peut amener à la création de quatre nouveaux CSP : dans le premier x a comme domaine $D = \{1\}$, dans le deuxième x a comme domaine $D = \{2\}$, dans le troisième x a comme domaine $D = \{3\}$ et dans le quatrième x a comme domaine $D = \{4\}$, i.e., dans chacun des quatre CSP la valeur de x est affectée.

La procédure `PropagationContraintes` est la procédure qui transforme le CSP en utilisant les méthodes de propagation de contraintes présentées à la section 2.4.3.

Après le partage du CSP $\mathcal{P}$ en nouveaux CSP $\mathcal{P}'_i$, l'ordre dans lequel ces nouveaux CSP sont considérés dépend de la technique de recherche utilisée. En fait, un arbre de CSP est créé par l'utilisation répétée de la procédure `Partager`. Les deux techniques les plus

utilisées pour parcourir cet arbre de CSP sont celles que nous avons décrites ci-dessus, i.e., la recherche avec retour-arrière (backtracking) et, dans le cas de problèmes d'optimisation sous contraintes, la recherche par séparation et évaluation progressive (branch and bound).

Nous allons maintenant décrire les contraintes globales AllDifferent et SomeDifferent.

2.4.5 Les contraintes globales AllDifferent et SomeDifferent

Une variation du problème de D-coloration de graphes présenté à la section 2.1.2 peut être utilisée pour résoudre une contrainte globale de CSP : la contrainte SomeDifferent. Cette contrainte est elle-même une variation d'une autre contrainte globale très connue : la contrainte AllDifferent. La contrainte `AllDifferent` [119, 136] impose que chaque paire de deux variables doit avoir des valeurs différentes et est présente dans de nombreux problèmes combinatoires.

Définition 2.4.11. Soit les variables $x_1, \ldots, x_n$ ayant les domaines respectifs $D_1, \ldots, D_n$. Alors la contrainte *AllDifferent* est définie de la façon suivante :

$$\texttt{AllDifferent}(x_1, \ldots, x_n) = \{(a_1, \ldots, a_n) \mid a_i \in D_i \text{ et } a_i \neq a_j \; \forall i \neq j\}.$$

Nous pouvons observer qu'une contrainte AllDifferent sur n variables est équivalente à $\frac{n(n-1)}{2}$ contraintes de non-égalité binaires. En effet, il y a une contrainte de non-égalité binaire $a_i \neq a_j$ pour chaque $i = 1, ..., n$ et chaque $j = i + 1, ..., n$.

Exemple 2.4.12. *Le problème des n reines peut se modéliser grâce à des contraintes AllDifferent. Ce problème consiste à placer n reines sur un échiquier de taille $n \times n$ (où $n \geq 3$) de telle sorte qu'aucune reine n'attaque une autre reine. Une première façon de modéliser ce problème consiste à introduire n variables entières $x_1, \ldots, x_n$, chaque x_i*

ayant le domaine $D_i = \{1, \ldots, n\}$. L'idée est que x_i détermine la position (= numéro de la ligne) de la reine placée dans la i-ème colonne de l'échiquier. Alors les contraintes de "non-attaque" peuvent s'exprimer de la façon suivante :

$$x_i \neq x_j \quad \text{pour } 1 \leq i < j \leq n, \tag{2.1}$$

$$x_i - x_j \neq i - j \quad \text{pour } 1 \leq i < j \leq n, \tag{2.2}$$

$$x_i - x_j \neq j - i \quad \text{pour } 1 \leq i < j \leq n, \tag{2.3}$$

$$x_i \in \{1, \ldots, n\} \quad \text{pour } 1 \leq i \leq n. \tag{2.4}$$

Les contraintes (2.1) imposent que deux reines ne soient pas dans la même ligne. Les contraintes (2.2) et (2.3) imposent que deux reines ne soient pas sur la même diagonale. Les contraintes (2.2) et (2.3) peuvent être réécrites de la façon suivante :

$$x_i - i \neq x_j - j \quad \text{pour } 1 \leq i < j \leq n, \tag{2.2}$$

$$x_i + i \neq x_j + j \quad \text{pour } 1 \leq i < j \leq n. \tag{2.3}$$

Ce qui nous permet d'utiliser des contraintes AllDifferent pour les contraintes de type (2.1), (2.2) et (2.3).

$$\texttt{AllDifferent}(x_1, \ldots, x_n), \tag{2.5}$$

$$\texttt{AllDifferent}(x_1 - 1, \ldots, x_n - n), \tag{2.6}$$

$$\texttt{AllDifferent}(x_1 + 1, \ldots, x_n + n), \tag{2.7}$$

$$x_i \in \{1, \ldots, n\} \text{ pour } 1 \leq i \leq n. \tag{2.8}$$

La contrainte AllDifferent est très importante en programmation par contraintes. C'est une des premières contraintes globales à avoir été décrite. De nombreux articles utilisent la contrainte AllDifferent pour montrer que son utilisation permet de résoudre plus rapidement certains problèmes. Cette contrainte permet de modéliser de nombreux

problèmes comme le décrit Van Hoeve dans son étude [136], par exemple pour la gestion du trafic aérien ou des problèmes d'horaires.

Parfois ce n'est pas chaque paire de variables qui doivent obtenir une valeur différente mais seulement un sous-ensemble de paires de variables. Richter et al. [122] ont étudié une telle contrainte globale et l'ont appelée `SomeDifferent`.

Définition 2.4.13. Soit les variables $x_1, \ldots, x_n \in \mathcal{X}$ ayant les domaines respectifs $D_1, \ldots, D_n$ et un graphe $G = (\mathcal{X}, E)$. Alors la contrainte `SomeDifferent` est définie de la façon suivante :

$$\texttt{SomeDifferent}(x_1, \ldots, x_n) = \{(a_1, \ldots, a_n) \mid a_i \in D_i \; \forall i \text{ et } a_i \neq a_j \; \forall (i,j) \in E\}.$$

La contrainte SomeDifferent peut être décrite en utilisant un modèle de D-coloration de graphe. En effet, soit un problème SomeDifferent décrit au moyen de l'ensemble de variables $\mathcal{X} = \{x_1, ..., x_n\}$ ayant les domaines $D = \{D_1, ..., D_n\}$, et d'un graphe $G = (\mathcal{X}, E)$. Il est possible de construire un graphe $G' = (V, E)$ avec l'ensemble de sommets $V = \{1, ..., n\}$ et l'ensemble d'arêtes E. Chaque sommet v de G' a le domaine D_v de couleurs possibles. Le problème SomeDifferent revient à chercher une D-coloration des sommets de G'.

Chaque arête $(u, v) \in E$ impose d'attribuer des valeurs distinctes pour u et v. Si G' est une clique, alors aucune paire de sommets ne peut avoir la même valeur, ce qui correspond à la contrainte usuelle `AllDifferent`. Sinon, seulement les paires de sommets reliés par une arête doivent avoir des valeurs distinctes ce qui correspond à la contrainte `SomeDifferent`, qui est donc plus générale.

Un exemple d'application pratique d'utilisation de la contrainte SomeDifferent est le problème de planification de la main-d'oeuvre comme le décrivent Richter et al. [122].

Étant donné une liste de tâches à effectuer (ayant chacune une date de début et de fin), une liste d'employés et une liste spécifiant quels employés peuvent effectuer chaque tâche, le but est de trouver une affectation telle que chaque tâche soit effectuée. Ceci peut être modélisé de la façon suivante : chaque tâche correspond à une variable et chaque employé est une couleur. Le domaine de chaque variable est formé par la liste des couleurs des employés qui peuvent effectuer la tâche correspondant à la variable. On ajoute une arête entre les variables correspondant aux tâches qui se chevauchent dans le temps. D'autres exemples d'applications pratiques de ce problème sont la conception de circuits, la création d'horaires d'examens ou la création d'horaires d'occupation de machines [122].

Dans le chapitre 6, nous présentons un algorithme de filtrage pour la contrainte SomeDifferent. Pour cette raison, nous précisons maintenant quelques notions dont nous avons besoin afin de présenter la cohérence de domaine de la contrainte SomeDifferent.

Définition 2.4.14. *Un couple sommet-couleur pour une contrainte SomeDifferent* est une paire (v, i) où $v \in V$ et $i \in D_v$.

Un couple sommet-couleur (v, i) est dit *supporté* dans G s'il existe une $D-$coloration c de G où le sommet v a la couleur $c(v) = i$. Si un couple sommet-couleur (v, i) n'est pas supporté dans G, alors la couleur i peut être supprimée du domaine D_v et on dit que le couple sommet-couleur peut être *filtré* de G. Notre but est d'obtenir la cohérence de domaine ce qui correspond à trouver de nouveaux domaines $D'_1, ..., D'_n$ de telle sorte que $i \in D'_v$ si et seulement si (v, i) est un couple sommet-couleur supporté dans G. Donc le but du filtrage est d'enlever toutes les valeurs $i \in D_v$ telles que (v, i) ne sont pas des couples sommet-couleur supportés. Ce problème est NP-difficile [122].

2.5 Sous-ensembles incohérents irréductibles

Il arrive fréquemment que, quand un CSP est non réalisable, ce ne sont pas toutes les variables ou toutes les contraintes qui sont impliquées dans l'incohérence mais qu'une partie d'entre elles explique déjà l'incohérence.

Définition 2.5.1. Un sous-ensemble de contraintes est *incohérent* ou *inconsistant* s'il est impossible de satisfaire toutes les contraintes de cet ensemble. Nous appellerons un tel ensemble un IS (pour *infeasible subset*) de contraintes.
Il est *incohérent minimal* (au sens de l'inclusion) ou *incohérent irréductible* s'il est non réalisable, mais devient réalisable dès qu'on lui supprime n'importe quelle contrainte. Nous appelons un tel sous-ensemble un IIS (pour *infeasible irreducible subset*) de contraintes et nous allons utiliser la notation IIS-C.

Selon le type de CSP, les IIS de contraintes peuvent porter des noms différents. Par exemple, pour le problème SAT, ils sont appelés MUSC (*minimum unsatisfiable subset of clauses*) par Desrosiers et al. [40]. Ceci va être précisé dans le chapitre 3.

Si nous reprenons l'exemple de la Figure 2.1 (page 8) et que nous essayons de trouver une 2-coloration, nous pouvons remarquer que ce problème est non réalisable. Il y a dans cet exemple deux sous-ensembles de contraintes qui forment des IIS de contraintes : le premier est formé par les arêtes du triangle (a, b), (a, f) et (b, f) et le deuxième est formé par les arêtes du pentagone (b, c), (c, d), (d, e), (e, f), (f, b).

Définition 2.5.2. Pour un CSP donné, une *affectation partielle* est une affectation d'une partie des variables (mais pas forcément de toutes), chaque valeur affectée à une variable faisant partie du domaine de la variable. Une affectation partielle est aussi appelée une *assignation partielle*, une *instanciation partielle* ou une *interprétation partielle*. Quand toutes les variables reçoivent une valeur, l'affectation est dite **complète**.

Définition 2.5.3. Une affectation partielle *satisfait une contrainte* C si elle peut être étendue à une affectation complète qui satisfait C.
Une affectation partielle est dite *légale* si elle satisfait toutes les contraintes du CSP.

Définition 2.5.4. Un sous-ensemble W de variables d'un CSP est *incohérent* s'il n'existe pas d'affectation partielle légale des variables de W, sinon il est dit *cohérent*. W est *incohérent minimal* (au sens de l'inclusion) ou *incohérent irréductible* s'il est incohérent mais devient cohérent dès que n'importe quelle variable de W est supprimée. Un tel sous-ensemble est appelé un IIS de variables et nous allons le noter IIS-V.

À nouveau, selon le type de CSP, les IIS de variables peuvent porter des noms différents. Si nous reprenons l'exemple de la Figure 2.1 et que nous cherchons une 2-coloration des sommets, ce problème n'est pas réalisable. Il y a deux IIS de variables : le premier est formé par les variables $\{a, b, f\}$ et le deuxième par les variables $\{b, c, d, e, f\}$.

Précisons la différence entre les termes *minimal* et *minimum* : minimal est équivalent à irréductible, i.e., qu'il n'est pas possible d'enlever une variable ou une contrainte sans rendre le sous-problème obtenu réalisable (et nous allons utiliser ces deux termes "minimal" et "irréductible" indifféremment) tandis que minimum signifie irréductible et de cardinalité minimum. De plus, nous allons dire qu'un sous-ensemble incohérent de contraintes ou variables est *minimisé* si des contraintes ou des variables lui sont enlevées afin de le rendre incohérent minimal.

La détection des IIS dans les CSP est un problème très intéressant. Les principaux avantages de cette détection sont :

- comprendre pourquoi le problème est non réalisable ;
- permettre de savoir quelles sont les contraintes à modifier dans le problème original pour obtenir un CSP réalisable ;
- permettre de démontrer plus facilement ou plus rapidement la non réalisabilité d'un

problème. En effet, si nous parvenons à démontrer la non réalisabilité d'un sous-problème, nous avons la preuve de la non réalisabilité du problème original. De plus, comme un IIS est généralement de taille plus petite que l'instance de départ, la preuve de la non réalisabilité devrait se faire plus rapidement (en pratique, cette dernière assertion n'est toutefois pas toujours vérifiée).

Dans ce travail, nous allons notamment présenter des techniques de détection automatique d'IIS de contraintes et de variables pour le problème SAT et pour un problème de confection d'horaires pour le personnel navigant aérien.

2.6 Conclusion

Ce chapitre nous a permis de définir formellement les différentes définitions et les différents problèmes qui sont utilisés dans cette thèse.

Les chapitres suivants sont groupés par deux (à chaque fois, le premier étant une revue de la littérature et le deuxième présentant les développements originaux effectués) et concernent les trois développements qui ont été faits dans le cadre de cette thèse :

- recherche d'IIS de contraintes et de variables pour le problème SAT,
- algorithme de filtrage pour la contrainte SomeDifferent,
- recherche d'IIS de contraintes pour le problème de confection d'horaires pour le personnel navigant aérien.

CHAPITRE 3

REVUE DE LA LITTÉRATURE CONCERNANT L'EXTRACTION D'IIS DANS LES CSP, LA RÉSOLUTION DU PROBLÈME SAT ET L'EXTRACTION D'IIS POUR LE PROBLÈME SAT

Dans ce chapitre, nous présentons d'abord une revue de la littérature concernant l'extraction d'IIS dans les CSP généraux et dans les problèmes de k-coloration de graphes. Puis, nous présentons une revue de la littérature des méthodes de résolution du problème SAT et décrivons les méthodes existantes pour l'extraction d'IIS de contraintes pour le problème SAT.

3.1 Détection de sous-ensembles incohérents dans des CSP

Nous présentons maintenant quelques méthodes générales de recherche d'IIS dans les CSP. Puis, nous présentons des méthodes qui ont été développées particulièrement pour le problème de k-coloration de graphes. Les méthodes de recherche d'IIS pour le problème SAT vont être présentées dans la section 3.2 qui est entièrement consacrée au problème SAT.

3.1.1 Extraction de sous-problèmes incohérents dans les CSP généraux

Dans cette section, nous présentons des techniques générales d'extraction d'IIS. Galinier et Hertz [48] présentent trois algorithmes exacts pour résoudre le problème de recouvrement de grands ensembles (de l'anglais *Large Set Covering Problem, LSCP*). Ils montrent comment l'extraction d'IIS de contraintes et variables dans des problèmes

de satisfaction de contraintes sont des cas particuliers du LSCP. Deux de leurs algorithmes permettent de trouver des sous-ensembles incohérents minimaux, ce sont les méthodes appelées `Removal` et `Insertion` et la troisième, `HittingSet`, trouve des sous-ensembles incohérents de cardinalité minimum. Comme nous allons adapter ces techniques pour l'extraction d'IIS de contraintes et de variables pour le problème SAT et l'extraction d'IIS de contraintes pour le problème de confection d'horaires, nous allons présenter en détail ces méthodes dans le Chapitre 4. Galinier et Hertz présentent aussi des versions heuristiques de ces trois algorithmes ainsi que plusieurs procédures permettant de trouver des bornes inférieures sur la taille des IIS.

Afin de pouvoir présenter des méthodes d'extraction d'IIS, nous introduisons d'abord quelques nouvelles définitions.

Définition 3.1.1. *Un sous-ensemble satisfaisable maximal, MSS (maximal satisfiable subset)* est un sous-ensemble réalisable de contraintes C' de $\mathcal{F}$ tel que si nous ajoutons n'importe quelle contrainte $C \in \mathcal{F} \setminus C'$ à C', alors la sous-formule obtenue est incohérente.

Toute solution du problème Max-SAT définit aussi un MSS. En effet, une solution de Max-SAT est de cardinalité maximum. Aucune autre contrainte ne peut être ajoutée à cette solution sans que la formule obtenue ne devienne non réalisable. Par contre, les MSS pouvant être de tailles différentes, ils ne sont pas tous des solutions de Max-SAT.

Définition 3.1.2. *Le complémentaire d'un MSS* C' est appelé un *coMSS* et $coMSS(C') = \overline{C'} = C \setminus C'$.

Définition 3.1.3. Soit S une collection de sous-ensembles d'un ensemble fini D. Un *hitting set* H de S est un sous-ensemble $H \subseteq D$ tel que $\forall\, S_i \in S$, $H \cap S_i \neq \emptyset$. Chaque ensemble S_i de S doit contenir au moins un élément de H. Un hitting set H est *minimal* ou *irréductible* si aucun sous-ensemble $h \subset H$ n'est un hitting set de S (aucun élément ne peut être enlevé à H sans qu'il ne perde sa propriété de hitting set).

Le problème de recherche d'un hitting set est NP-difficile [55].

Afin de trouver tous les IIS-C de systèmes de contraintes de Herbrand utilisées en "correction d'erreurs (error type debugging)" et "détection d'erreurs (error finding)", Bailey et Stuckey [11] adaptent un algorithme nommé *Dualize and Advance*, DAA, qui a été proposé auparavant par Gunopulos et al. [71] dans le cadre du Data Mining pour trouver des collections de motifs fréquents maximaux. L'idée de leur méthode se base sur la dualité entre les IIS-C et les *MSS*. En effet, si nous notons Q l'ensemble de tous les *MSS* d'une formule $\mathcal{F}$, alors $\overline{Q}$ est l'ensemble de tous les *coMSS*, et l'ensemble de tous les IIS-C de $\mathcal{F}$ est trouvé en cherchant tous les hitting sets de $\overline{Q}$. Bailey et Stuckey [11] donnent l'exemple suivant.

Exemple 3.1.4. *Supposons qu'un problème $\mathcal{P}$ a quatre contraintes $C = \{C_1, C_2, C_3, C_4\}$ et supposons que l'ensemble Q des MSS est $Q = \{\{C_3\}, \{C_4\}, \{C_2\}\}$. Alors l'ensemble des coMMSs est $\overline{Q} = \{\{C_1, C_2, C_4\}, \{C_1, C_2, C_3\}, \{C_1, C_3, C_4\}\}$ et les IIS-C sont les hitting sets minimaux de $\overline{Q}$, i.e., $\{\{C_1\}, \{C_2, C_4\}, \{C_2, C_3\}, \{C_3, C_4\}\}$.*

DAA construit les *MSS* de manière directe en les augmentant. Il part d'un ensemble de contraintes X réalisables (cet ensemble X est vide à la première itération). Un *MSS* M est construit en ajoutant successivement chacune des contraintes restantes de la formule à cet ensemble X et en gardant seulement les contraintes qui ne créent pas de conflit. L'ensemble M ainsi construit est bien un *MSS* car les contraintes non ajoutées sont celles qui auraient rendu le problème non réalisable.
Chaque fois qu'un *MSS* est trouvé de cette façon, son complément (le *coMSS*) est ajouté à l'ensemble augmentant des *coMSS*. À cette étape, les hitting sets de l'ensemble des *coMSS* sont calculés pour obtenir des IIS-C candidats et/ou obtenir un nouvel ensemble de départ X (pour la recherche de *MSS*). Chaque hitting set qui est non réalisable est un IIS-C. Dès qu'un des hittings sets trouvés est réalisable alors celui-ci est utilisé comme nouvel ensemble de départ X pour la prochaine itération de recherche d'un *MSS* et l'algorithme passe à la prochaine itération. L'ensemble de départ X ainsi créé est assuré de

n'avoir aucune intersection commune avec les précédents *MSS* trouvés parce qu'il a été créé à partir des coMSS (pour chaque MSS trouvé auparavant le nouvel MSS contiendra au moins une contrainte qui n'en faisait pas partie). Quand à une itération, aucun des hitting sets trouvés n'est réalisable, l'algorithme s'arrête.
La recherche des hitting sets peut se faire de plusieurs façons. Ce problème est aussi connu sous le nom *hypergraph transversal problem*. Bailey et Stuckey ont implémenté une façon simple de trouver les hitting sets de Q s'inspirant de Berge [20].

Dans le contexte de la programmation linéaire, Amaldi et al. [6, 7] et Gleeson et Ryan [61] ont proposé des méthodes pour l'extraction d'IIS de contraintes.
Dans le cas de systèmes d'inégalités linéaires avec variables à valeurs entières, Chinneck [31] a proposé des méthodes additives et soustractives pour trouver des IIS de contraintes. De plus, Chinneck [31] donne une revue de littérature étendue des autres méthodes existantes pour extraire des IIS de contraintes en programmation linéaire.

Grégoire et al. [68] et Hemery et al. [73] appellent les IIS de contraintes dans les CSP des MUC (pour *minimally unsatisfiable cores*).
Hemery et al. [73] proposent un algorithme pour extraire un IIS-C qui se déroule en deux phases : *wcore* et *DC(wcore)*. Pendant la première phase, une procédure appelée *wcore* extrait un sous-problème non réalisable (pas forcément minimal). Ensuite, une deuxième phase supprime des contraintes du sous-problème non réalisable obtenu précédemment afin de le rendre irréductible (c'est la phase dite de minimisation). La première phase se base sur le fait que quand un algorithme de branchement prouve qu'un CSP est non réalisable, il peut en extraire un sous-problème non réalisable (montré par Bakker et al. [12]). Ceci se fait grâce aux contraintes dites *actives*. Étant donnée une affectation partielle des variables d'un CSP, une contrainte C est dite *active* si lors de l'opération de maintien de cohérence d'arc (MAC vu à la page 27), C a provoqué par arc-cohérence, le retrait d'une valeur du domaine d'une variable du CSP. Donc si une

affectation partielle des variables de $\mathcal{P}$ est effectuée, les contraintes actives sont celles qui ont permis de réduire l'espace des solutions (car elles ont enlevé des valeurs dans les domaines des variables). Lorsqu'une recherche complète d'une solution est effectuée avec un algorithme de branchement, différentes affectations partielles sont rencontrées. C'est l'union des contraintes actives dans ces affectations partielles qui permet de déduire un sous-problème incohérent (pas minimal). Donc l'utilisation d'une recherche complète prouvant la non réalisabilité d'un CSP suffit à trouver un IS de contraintes du CSP. Afin de trouver ce sous-ensemble incohérent, il suffit de marquer les contraintes actives durant la recherche complète. En effet, les autres contraintes (contraintes non actives) ne participent pas à la preuve de non réalisabilité car elles n'ont pas réduit le domaine d'une variable. Donc, ces contraintes non actives peuvent être supprimées du problème et le sous-problème obtenu est toujours non réalisable. L'efficacité de cette méthode dépend du choix des variables affectées à chaque étape du branchement. Pour cela les auteurs utilisent l'heuristique *dom/wdeg* de choix de la variable de branchement de Boussemart et al. [24]. Cette heuristique associe un compteur à chaque contrainte C du problème. Ces compteurs permettent de pondérer les contraintes. Quand, lors de la propagation de contraintes, une contrainte est montrée insatisfaite, son poids est augmenté de 1. Le poids *wdeg* d'une variable X est défini comme la somme des poids des contraintes agissant sur X et sur au moins une variable non affectée. Ensuite, pour chaque variable X le ratio *dom/wdeg* est calculé où *dom* est la taille courante du domaine de X, et l'heuristique choisit la variable avec le plus petit ratio *dom/wdeg*. Cette technique d'extraction de sous-problème non réalisable peut être itérée pour extraire un nouveau sous-problème incohérent. Pour la deuxième phase, ils proposent trois techniques de minimisation : une constructive, une destructive et une dichotomique. Pour chacune de ces trois méthodes, ils déterminent la contrainte C_i telle que le CSP formé par $\{C_1, \ldots, C_{i-1}\}$ est réalisable tandis que le CSP formé par $\{C_1, \ldots, C_i\}$ est non réalisable. Cette contrainte C_i est appelée *contrainte de transition*. Toutes les contraintes C_j avec $j > i$ peuvent alors être enlevées du CSP. Au départ, un ordre de traitement des

contraintes est fixé. L'approche constructive détecte une contrainte de transition en ajoutant successivement des contraintes à un CSP courant (au départ vide) jusqu'à ce que le CSP devienne non réalisable. L'approche destructive détecte une contrainte de transition en partant du CSP courant (au départ le CSP original) et en supprimant successivement des contraintes jusqu'à ce que le CSP devienne réalisable. L'approche dichotomique détecte une contrainte de transition en utilisant une recherche dichotomique. En utilisant une des méthodes décrites ci-dessus, une première contrainte de transition C_i est trouvée. Toutes les contraintes C_j avec $j > i$ sont supprimées ce qui permet d'obtenir un nouveau CSP $\mathcal{P}'$. Ensuite, un nouvel ordre de traitement des contraintes est fixé en s'assurant que C_i est le dernier élément de cet ordre et le processus de recherche recommence jusqu'à ce que toutes les contraintes restantes soient des contraintes de transition. Grégoire et al. [68] proposent une amélioration de la première étape.

3.1.2 Détection des sous-ensembles incohérents dans le problème de k-coloration de graphe

Dans cette section, nous présentons des méthodes d'extraction de sous-problèmes incohérents minimaux pour le problème de k-coloration de graphe.

Définition 3.1.5. Un graphe G est *sommet-critique* si $\chi(H) < \chi(G)$ pour chaque sous-graphe $H \subset G$ obtenu en enlevant n'importe quel sommet de G. De même, G est *arête-critique* si le fait d'enlever n'importe quelle arête de G diminue strictement $\chi(G)$.

Définition 3.1.6. Étant donné un entier $k \in \mathbb{N}$ un *sous-graphe k-sommet-critique de G* est un sous-graphe sommet-critique $G' \subseteq G$ tel que $\chi(G') = k$. De même, un *sous-graphe k-arête-critique de G* est un sous-graphe arête-critique $G' \subseteq G$ tel que $\chi(G') = k$.

Définition 3.1.7. Un sous-graphe k-arête-critique (respectivement k-sommet-critique) est *minimum* si G n'a pas d'autre sous-graphe critique contenant moins d'arêtes (resp. sommets).

Donc en fait, un sous-graphe k-arête-critique (resp. k-sommet-critique) de G est un IIS de contraintes (resp. variables) pour le CSP qui correspond à trouver une $(k-1)$-coloration de G.

Comme nous l'avons précisé dans le chapitre 2, trouver le nombre chromatique $\chi(G)$ d'un graphe G est NP-difficile [55]. Donc, les algorithmes exacts développés pour ce problème [25,88,104,115] ne peuvent être appliqués que sur des petites instances. Il est possible d'utiliser des heuristiques [44,47] pour les plus grands problèmes, mais ceux-ci ne fournissent que des bornes supérieures sur $\chi(G)$.

Herrmann et Hertz [74] proposent un nouvel algorithme exact qui utilise cette notion de sous-graphe critique pour calculer le nombre chromatique d'un graphe G. Leur algorithme combine l'utilisation d'une méthode exacte, *ExactCol*, et d'une méthode heuristique, *HeurCol*, pour la détermination du nombre chromatique. Ils précisent les notations suivantes. Si $G = (V, E)$ est un graphe et si $x \in V$ est un sommet de G, alors $G - x$ est le sous-graphe de G induit par $V \setminus \{x\}$. Si $G' = (V', E')$ est un sous-graphe de $G = (V, E)$ et si $x \in V \setminus V'$, alors $G' + x$ est le sous-graphe de G induit par $V' \cup \{x\}$. Leur algorithme fonctionne de la façon suivante. Dans une phase d'initialisation, *HeurCol* calcule une borne supérieure k de $\chi(G)$. Ensuite une phase descendante est effectuée. Celle-ci a pour but de trouver un sous-graphe induit critique G' de G tel que $\chi(G') = \chi(G)$ et fonctionne comme suit. Au départ G' est posé égal à G, puis les sommets x de G' sont retirés successivement selon un ordre fixé et lors de chaque retrait *HeurCol* calcule une borne supérieure sur $\chi(G' - x)$ et compare si c'est égal à la borne supérieure obtenue sur $\chi(G')$. Si c'est bien le cas, la suppression continue avec le sommet suivant dans l'ordre fixé. Sinon le sommet x est réintroduit dans G' et la suppression continue avec le sommet suivant. Quand tous les sommets ont été testés alors *ExactCol* calcule le nombre chromatique exact $\chi(G')$ du graphe G' actuel. Si $\chi(G') = k$, alors G' est un sous-graphe critique de G et $\chi(G') = \chi(G)$ et l'algorithme s'arrête. Il peut arriver que $\chi(G') < k$. Dans de tels cas, une phase augmentante débute dans laquelle des sommets sont ajou-

tés à G' jusqu'à ce que soit $\chi(G') = k$ soit $G' = G$. Afin de faire cela, une liste L de sommets est créée. Au départ L est vide, puis sont ajoutés à L les sommets x de $G \setminus G'$ (i.e., les sommets retirés de G') tels que la borne supérieure k' calculée par *HeurCol* sur $\chi(G' + x)$ soit strictement plus grande que la valeur exacte $\chi(G')$. Si à la fin de la phase augmentante L n'est pas vide, alors un sommet de L est choisi et ajouté à G', ce qui fait que $\chi(G')$ augmente de 1. Ce processus est ensuite répété soit jusqu'à ce que L soit vide, soit jusqu'à ce que $\chi(G') = k$. Afin de trouver les sous-graphes critiques les plus petits possibles, Herrmann et Hertz proposent que l'ordre de traitement des sommets dans la phase descendante corresponde avec l'ordre ascendant des degrés des sommets de G (i.e., d'abord sont supprimés les sommets ayant le moins de sommets adjacents).

Desrosiers et al. [39] ont adapté les techniques `Removal`, `Insertion` et `Hitting-Set` présentées par Galinier et Hertz [48] afin de trouver des sous-graphes arêtes-critiques et sommets-critiques d'un graphe donné G et montrent comment ces techniques peuvent être utilisées pour déterminer le nombre chromatique de G. Comme nous allons aussi utiliser ces trois techniques pour trouver des IIS de contraintes et de variables pour le problème SAT et des IIS de contraintes pour le problème d'horaire de personnel navigant, nous allons présenter ces méthodes en détail dans la Section 4.1. La méthode `Removal` est semblable à celle présentée ci-dessus par Herrmann et Hertz. Desrosiers et al. utilisent des heuristiques de recherche tabou afin d'estimer le nombre chromatique. De plus, ils présentent un algorithme permettant de calculer des bornes inférieures sur le nombre chromatique d'un graphe G.

3.2 Le problème SAT et sa résolution

3.2.1 Algorithmes de résolution du problème SAT

Avant de présenter les méthodes d'extraction de sous-formules incohérentes minimales pour le problème SAT, nous allons présenter différents algorithmes de résolution du problème SAT.
Il existe deux grandes catégories d'algorithmes de résolution du problème SAT : les algorithmes complets et les algorithmes incomplets.

3.2.1.1 Algorithmes complets

Ces algorithmes vont trouver une solution de façon certaine ou démontrer qu'il n'en existe pas. Pour les formules normales conjonctives non réalisables, ce sont les seuls algorithmes qui peuvent répondre à cette question de manière exacte. Afin d'y parvenir, ils parcourent un arbre de recherche avec retour-arrière (présentés de façon générale à la page 28). Comme dans le cas du problème SAT les domaines des variables sont tous de taille 2 (seules les valeurs vrai=1 ou faux=0 sont possibles), à chaque noeud de l'arbre, il y a deux nouvelles branches possibles. C'est pour cette raison que, dans ce cas-ci, ces arbres sont appelés des arbres binaires de recherche. Comme nous l'avons vu, nous appelons un noeud de l'arbre un problème où une variable reçoit une affectation donnée. Si le problème a n variables, alors il y a 2^n noeuds possibles dans l'arbre de recherche binaire. À chaque noeud, il faut vérifier s'il vaut la peine d'explorer les branches en-dessous, donc il faut évaluer la cohérence de l'affectation partielle. Or comme le nombre de noeuds total possible est 2^n, cela vaut la peine de pouvoir couper le plus vite possible les branches aboutissant à des solutions non réalisables. C'est pour cela que la plupart des algorithmes sont basés sur le principe de l'*échec d'abord* (de l'anglais *First Fail*) :

le but est de trouver rapidement une contradiction dans les branches aboutissant à des affectations non réalisables. Pour y parvenir, deux "outils" sont principalement utilisés dans les algorithmes complets : ce sont la *saturation* et la *simplification*.

La *saturation* est aussi appelée la *résolution*. Le principe de *saturation* [18] consiste à générer des nouvelles clauses (appelées *résolvantes*) soit jusqu'à l'obtention d'une clause vide (signifiant que la formule est non réalisable) soit jusqu'à ce qu'aucune nouvelle clause résolvante ne puisse être créée.

La saturation agit sur deux clauses $C_1 = z \vee A$ et $C_2 = \bar{z} \vee B$, ayant une variable z en commun : C_1 contient le littéral positif z et C_2 contient le littéral négatif $\bar{z}$. La saturation réunit les deux parties différentes A et B pour créer une nouvelle clause $C_3 = A \vee B$ (qui ne modifie pas la satisfaisabilité du problème initial).

Exemple 3.2.1.
Prenons comme exemple, $\mathcal{F}_1$.

$$\begin{aligned}
\mathcal{X} &= \{x_1, x_2, x_3, x_4, x_5, x_6, x_7\} \\
\mathcal{C} &= \{C_1, C_2, C_3, C_4, C_5\} \\
\mathcal{F}_1 &= C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5 \\
&\text{où} \\
C_1 &= x_1 \\
C_2 &= x_2 \vee x_3 \vee x_4 \\
C_3 &= \bar{x}_2 \vee x_5 \vee \bar{x}_6 \\
C_4 &= x_1 \vee x_7 \\
C_5 &= \bar{x}_1 \vee x_6
\end{aligned}$$

C_2 et C_3 ont la variable x_2 en commun (avec les deux littéraux opposés x_2 dans C_2 et $\bar{x}_2$ dans C_3) et C_4 et C_5 ont la variable x_1 en commun (avec les deux littéraux opposés

x_1 dans C_4 et $\bar{x}_1$ dans C_5).

La *saturation* va créer les deux nouvelles clauses suivantes.

À partir de C_2 et C_3 :

$$C_6 = x_3 \vee x_4 \vee x_5 \vee \bar{x}_6$$

Et à partir de C_4 et C_5 :

$$C_7 = x_7 \vee x_6$$

Et finalement à partir de C_6 et C_7 (variable x_6 en commun) :

$$C_8 = x_3 \vee x_4 \vee x_5 \vee x_7$$

Exemple 3.2.2.

Si nous prenons comme autre exemple, la formule $\mathcal{F}_2$ suivante :

$$\begin{aligned} \mathcal{F}_2 &= C_1 \wedge C_2 \\ \text{où} & \\ C_1 &= x_1 \\ C_2 &= \bar{x}_1 \end{aligned}$$

La saturation crée la clause $C_3 = \emptyset$, car C_1 contient le littéral positif x_1 tandis que C_2 contient son opposé. Ainsi, on aboutit à une clause vide ce qui montre que la formule est non réalisable.

Pour certaines formules, l'ordre dans lequel les clauses sont saturées peut avoir un effet sur l'effort demandé pour les résoudre.

Trois techniques de *simplification* sont principalement utilisées.

La première consiste à propager les clauses composées d'un unique littéral ([92] et

[18]) et est appelée `propagation des clauses unitaires` ou plus simplement `propagation unitaire`. Pour que la formule soit vérifiée, il y a une seule affectation possible pour les clauses composées d'un seul littéral. Ensuite, cette valeur est propagée aux autres clauses, i.e., toutes les clauses contenant le littéral sont supprimées car elles sont vérifiées et toutes les fois où apparaît le littéral opposé dans une clause, celui-ci est supprimé de la clause (mais la clause est conservée).
Reprenons l'exemple 3.2.1 (avec les clauses obtenues par saturation) : la clause C_1 implique que x_1 doit être fixée à *vraie*. Ainsi, par propagation, C_4 est vérifiée et C_5 devient $C_5 = x_6$, donc C_5 est une nouvelle clause unitaire.
Ainsi, x_6 doit être fixée à *vraie* et $C_3 = \bar{x}_2 \vee x_5$ et $C_6 = x_3 \vee x_4 \vee x_5$ et C_7 est vérifiée (car $C_7 = x_7 \vee x_6$).

Si une formule $\mathcal{F}$ contient deux clauses C_1 et C_2 telles que C_1 est incluse dans C_2, alors la clause C_2 peut être supprimée. Cette technique est parfois appelée la *subsomption*.

Exemple 3.2.3.
Pour illustrer cette deuxième technique, considérons que l'on a une formule $\mathcal{F}_3$ qui comprend deux clauses.

$$\begin{aligned} \mathcal{X} &= \{x_1, x_2, x_3, x_4\} \\ \mathcal{C} &= \{C_1, C_2\} \\ \mathcal{F}_3 &= C_1 \wedge C_2 \\ C_1 &= x_1 \vee x_2 \\ C_2 &= x_1 \vee x_2 \vee x_3 \vee \bar{x}_4 \end{aligned}$$

Comme C_1 est incluse dans C_2, C_2 peut être supprimée car dès que C_1 est satisfaite, C_2 est aussi satisfaite.

Cette technique est très rarement utilisée en pratique, car elle trop coûteuse.

La troisième technique de simplification est la *règle du littéral pur* [38] (aussi connue

sous le nom de la "règle positive-négative"). Afin de pouvoir la définir nous allons préciser la notation suivante. Soit $\mathcal{F}$ une formule normale conjonctive, alors $(\mathcal{F},\ x = 1)$ est la formule $\mathcal{F}$ dans laquelle la variable x est fixée à vraie dans toutes les clauses.
Un littéral est pur si toutes ses occurrences dans $\mathcal{F}$ sont soit toutes positives soit toutes négatives. Si seul le littéral positif x apparaît dans les clauses de $\mathcal{F}$, alors on peut remplacer $\mathcal{F}$ par $(\mathcal{F},\ x = 1)$ et supprimer les clauses ainsi vérifiées. De même, si seul le littéral négatif, $\bar{x}$, apparaît dans les clauses de $\mathcal{F}$, alors on peut remplacer $\mathcal{F}$ par $(\mathcal{F},\ x = 0)$ et supprimer les clauses ainsi vérifiées. Dans les deux cas, le littéral en question est dit *pur*.

Exemple 3.2.4.
Cet exemple illustre la règle du littéral pur.

$$\begin{aligned}
\mathcal{X} &= \{x_1, x_2, x_3, x_4, x_5\} \\
\mathcal{C} &= \{C_1, C_2, C_3, C_4\} \\
\mathcal{F}_3 &= C_1 \wedge C_2 \wedge C_3 \wedge C_4 \\
C_1 &= x_1 \vee x_2 \vee x_3 \\
C_2 &= \bar{x}_2 \vee x_3 \vee x_4 \\
C_3 &= \bar{x}_1 \vee \bar{x}_3 \\
C_4 &= x_5 \vee x_3 \vee \bar{x}_4
\end{aligned}$$

Le littéral x_5 apparaît une seule fois dans $\mathcal{F}_3$ comme un littéral positif. On le fixe à vrai, $x_5 = 1$. Ainsi, C_4 est vérifiée.
$\mathcal{F}_3$ devient : $\mathcal{F}_3 = C_1 \wedge C_2 \wedge C_3$.
Le littéral x_4 apparaît alors une seule fois (dans C_2), comme littéral positif. On fixe x_4 à vrai, $x_4 = 1$. Ainsi, C_2 est vérifiée.
$\mathcal{F}_3$ devient : $\mathcal{F}_3 = C_1 \wedge C_3$.
Le littéral x_2 devient alors un littéral pur positif. On fixe x_2 à vrai, $x_2 = 1$ et C_1 est vérifiée.
$\mathcal{F}_3$ devient : $\mathcal{F}_3 = C_3$.

Il suffit donc de fixer x_1 ou x_3 à faux pour pouvoir satisfaire $\mathcal{F}$.

La règle du littéral pur étant aussi très coûteuse, elle n'est pas souvent utilisée en pratique.

Le premier algorithme complet a été décrit par Davis et Putnam en 1960 [38]. Il utilisait les techniques décrites ci-dessus. À chaque étape, des clauses sont ajoutées par *saturation* et des clauses sont supprimées par *simplification*. Le désavantage est que le nombre de clauses ajoutées peut rapidement augmenter. En effet, on ajoute à une formule $\mathcal{F}$ tous les contraintes résolvantes possibles obtenues à partir de clauses faisant intervenir x_i et $\bar{x}_i$. Si x_i apparaît dans k clauses et $\bar{x}_i$ dans l clauses, on crée $k \cdot l$ nouvelles clauses et on en supprime $k + l$.

La procédure de Davis et Putnam, notée DP($\mathcal{F}$), est décrite dans la Figure 3.1. Si la résolution s'effectue sur la variable x_i, la procédure de saturation va produire des nouvelles clauses qui ne contiennent pas x_i. Ensuite, toutes les clauses contenant x_i sont supprimées de $\mathcal{F}$. Le résultat final est une clause vide si et seulement si l'étape de résolution produit une clause vide. Donc, si l'algorithme produit une clause vide, le résultat retourné est NON RÉALISABLE.

Dans cet algorithme, ChoixProchaineVariable() est une procédure qui choisit la prochaine variable traitée. Après l'algorithme 3.3 sont présentés des heuristiques de choix de la prochaine variable. `Saturation`(C_a, C_b) est la fonction qui effectue la saturation telle que décrite précédemment.

Le principal désavantage de cet algorithme est, qu'en plus d'avoir une complexité exponentielle en termes de temps de calcul (SAT est NP-complet), le nombre de clauses résolvantes peut être exponentiel lui aussi.

Ensuite, Davis avec Logemann et Loveland, [37], ont étendu ces recherches pour ob-

DP($\mathcal{F}$,$\mathcal{X}$)

Entrée : une formule CNF $\mathcal{F}$ ayant l'ensemble de variables $\mathcal{X}$
Sortie : la réponse NON RÉALISABLE (si la saturation crée une clause vide) ou RÉALISABLE

si $\mathcal{F} = \emptyset$ **alors retourner** *RÉALISABLE*;
si $\mathcal{F}$ *contient une clause vide* **alors retourner** *NON RÉALISABLE*;
$x \leftarrow$ `ChoixProchaineVariable()`;
Résolvantes $\leftarrow \emptyset$;
pour chaque (C_a, C_b) *tq* $C_a \in \mathcal{F}$ *et* $C_b \in \mathcal{F}$ *et* $x \in C_a$ *et* $\bar{x} \in C_b$ **faire**
  Résolvantes$\leftarrow$Résolvantes $\cup$ `Saturation` (C_a, C_b);
$C_x \leftarrow \{C \in \mathcal{F} | \ x \in C\}$;
$\mathcal{F} \leftarrow \mathcal{F} - C_x$;
$\mathcal{F} \leftarrow \mathcal{F} \cup$ Résolvantes;
$\mathcal{X} \leftarrow \mathcal{X} \setminus \{x\}$;
retourner $DP(\mathcal{F}, \mathcal{X})$;

Figure 3.1 – Procédure DP($\mathcal{F}$,$\mathcal{X}$) originale

tenir des techniques encore plus performantes. Ces techniques sont aujourd'hui appelées *Procédures DLL* ou *DPLL* . Les procédures DPLL construisent un arbre binaire de recherche, chaque appel récursif constituant un noeud de l'arbre. À chaque étape, l'algorithme sépare le problème en deux sous-problèmes. Le branchement se fait sur une variable x_i, dans une branche on affecte x_i à *vrai* et dans l'autre à *faux.* Pour chaque noeud, cela donne deux sous-problèmes, appelés problèmes enfants. On applique ensuite à chacun de ces deux enfants la propagation des clauses unitaires. Le parcours de cet arbre se fait en profondeur. À la racine se trouve la formule conjonctive de départ. À chaque noeud, se trouve une formule conjonctive résultante de l'affectation successive des variables. On doit stocker le chemin parcouru afin de faire un retour en arrière *(backtrack)* pour explorer d'autres branches. Si, pendant le parcours, on arrive à un noeud où la résultante donne un ensemble vide de clauses, alors on a une solution de la formule (il suffit de parcourir les chemins à l'envers). Par contre, si, à un noeud, on trouve une clause contredite, alors la branche peut être abandonnée.

La procédure DPLL telle qu'elle est le plus souvent utilisée (i.e., avec la propagation unitaire, mais sans la subsomption, la règle du littéral pur ni la saturation) est décrite dans la Figure 3.2. Cette procédure utilise la procédure de Simplification décrite dans la Figure 3.3.

DPLL

Entrée : Une formule normale conjonctive
Sortie : La réponse RÉALISABLE ou NON RÉALISABLE

si $\mathcal{F} = \emptyset$ **alors**
  retourner *RÉALISABLE*;
si $\mathcal{F}$ *contient une clause vide* **alors**
  retourner *NON RÉALISABLE*;
si $\mathcal{F}$ *contient une clause unitaire* $C_k = l$ **alors**
  retourner `DPLL` *(*`Simplification` *(*$\mathcal{F}$*,*l*))*;

$x \leftarrow$`ChoixProchaineVariable ()`
si `DPLL` *(*$\mathcal{F}, \{x = 1\}$*)* = *RÉALISABLE* **alors**
  retourner *RÉALISABLE*;
sinon
  retourner `DPLL` *(*$\mathcal{F}, \{x = 0\}$*)*;

Figure 3.2 – Procédure DPLL($\mathcal{F}$)

Le point essentiel de cette procédure DPLL est le choix de la variable sur laquelle on va brancher (`ChoixProchaineVariable()`). C'est ce facteur-là qui va déterminer la performance de l'algorithme.

Les deux types principaux d'heuristiques pour le choix de la variable de branchement sont : MOMS (*Maximum Occurrences in Minimum Size clauses*) [46, 123], et la propagation unitaire (*UP, Unit Propagation*) [33, 46, 92]. Ces deux techniques sont décrites

Simplification($\mathcal{F}$,l)

Entrée : Une formule normale conjonctive $\mathcal{F}$ ainsi qu'un littéral l
Sortie : La formule $\mathcal{F}$ simplifiée

Affecter la valeur de vérité à l afin de satisfaire $\{l\}$ et simplifier $\mathcal{F}$ en supprimant les clauses contenant l et en supprimant $\bar{l}$ des clauses le contenant;
retourner $\mathcal{F}$ *simplifiée*;

Figure 3.3 – Simplification($\mathcal{F}, l$)

ci-dessous.

MOMS (Maximum Occurrences in Minimum Size clauses) [46, 123]

Avec cette technique, on cherche à brancher sur une variable apparaissant le plus souvent dans les clauses les plus courtes. En effet, intuitivement, les littéraux appartenant aux clauses les plus courtes sont les littéraux les plus contraints de la formule. Un branchement sur ces littéraux devrait maximiser l'effet de la propagation unitaire (voir paragraphe suivant) et la vraisemblance d'atteindre une contradiction rapidement dans l'arbre de recherche pour les formules non réalisables. Plus précisément, notons

- $f_n(l)$ le nombre d'occurrences du littéral l dans les clauses de longueur n,
- $f^*(l)$ le nombre d'occurrences du littéral l dans les plus courtes clauses. En pratique, $f^*(l)$ est généralement remplacé par $f_2(l)$.

Alors, le but de l'heuristique est de choisir la variable l telle que $\min\{f^*(l), f^*(\bar{l})\}$ est maximum.

Pour y arriver, le score de chaque variable l est calculé de la façon suivante (si le but est de maximiser les variables apparaissant le plus dans des clauses binaires) :

$$s(l) = (f_2(l) \cdot f_2(\bar{l})) * 2^K + (f_2(l) + f_2(\bar{l}))$$

Le but de multiplier le premier terme de la somme par 2^K est de forcer la préférence pour les variables l telles que les valeurs $f_2(l)$ et $f_2(\bar{l})$ sont non-nulles et que ces deux valeurs sont à peu près égales. Les autres termes de la somme servent à trier les variables où une de ces deux valeurs est 0. La variable K est fixée arbitrairement grande au début et est diminuée s'il y a un débordement de mémoire. On branche sur la variable obtenant le plus grand score.

La raison pour laquelle cette heuristique fonctionne bien est le fait que les variables choisies pour le branchement sont les variables apparaissant le plus dans les clauses les plus contraintes (car elles sont de plus petite taille).

Cette méthode présente deux désavantages. Le premier est que l'efficacité de cette heuristique dépend considérablement du nombre de clauses binaires dans la formule normale conjonctive donnée. En revanche, cette heuristique fonctionne en général bien sur les problèmes réels, car ceux-ci ont tendance à contenir une large proportion de clauses binaires (montré par Freeman [46]).

Le deuxième désavantage est qu'il guide peu la recherche pour des formules contenant des clauses de même longueur, i.e., des *k-SAT* où $k \neq 1$ ou $k \neq 2$.

MOMS donne donc une approximation facile mais non exacte du nombre de propagations unitaires qu'une affectation particulière d'une variable peut produire.

Propagation unitaire (*UP heuristic*) [33, 46, 92]

Alternativement, on peut utiliser une heuristique de propagation unitaire (abrégée heuristique UP) qui calcule le nombre exact de propagations qui seraient causées par l'affectation particulière d'une variable.

La propagation unitaire est aussi appelée *Boolean Constraint Propagation (BCP)*.

Étant donné une variable x, une heuristique UP examine x en ajoutant respectivement la clause unitaire x et $\bar{x}$ à la formule SAT et fait indépendamment deux propagations unitaires. Le nombre précis de propagations unitaires produites est ensuite utilisé pour évaluer la variable de branchement. Contrairement à MOMS, cette heuristique évalue le

nombre exact de propagations unitaires engendrées par une affectation d'une variable. Le principal désavantage de cette heuristique est le fait qu'elle est très coûteuse car elle évalue chaque variable restant à estimer à chaque noeud de l'arbre de séparation et évaluation progressive. De ce fait, cette heuristique est souvent combinée avec une heuristique MOMS pour pouvoir optimiser la propagation unitaire qui va découler du choix de la variable, [46], [33]. MOMS est utilisée pour choisir un petit nombre de variables candidates, chacune d'entre elles étant ensuite évaluée de façon exacte en utilisant l'heuristique de propagation (qui est plus coûteuse).
Les algorithmes Satz de Li et Anbulagan [92], et zChaff de Moskewicz et al. [107] fonctionnent de cette manière.

Parmi les autres heuristiques de choix de la variable de branchement, citons encore la méthode de Jeroslow et Wang [81]. Ils estiment la contribution que chaque littéral pourrait avoir dans le processus de propagation unitaire de la façon suivante. Le score de chaque littéral l est le suivant : $score(l) = \sum_{l \in \text{ contrainte } C} 2^{-|C|}$. Le branchement est ensuite effectué sur le littéral qui a le plus grand score.

3.2.1.2 Algorithmes incomplets

Un algorithme incomplet est un algorithme qui ne parcourt pas tout l'espace de recherche. Il est possible qu'un tel algorithme ne trouve pas de solution, même s'il en existe une. Les algorithmes incomplets sont basés principalement sur la recherche locale ou sur l'utilisation des probabilités.

3.2.1.2.1 Méthodes de type PPSZ

Ces méthodes utilisent les probabilités pour résoudre des formules *k-SAT* réalisables. La plupart de ces algorithmes se concentrent sur les *3-SAT*.
En 1988, Paturi et al. [113] proposent un algorithme aléatoire pour le problème SAT. Soit $\mathcal{F}$ une formule normale conjonctive et $\mathcal{X}$ un ensemble de n variables. Leur algorithme se déroule en deux étapes.

1. Une première étape de prétraitement applique le principe de saturation pour augmenter le nombre de clauses de $\mathcal{F}$. La longueur de chaque clause générée ne peut excéder une limite fixée.

2. Ensuite, une étape de recherche utilise une procédure aléatoire gloutonne pour chercher une affectation des variables satisfaisant la formule $\mathcal{F}$.
   Les variables sont traitées selon un ordre π déterminé aléatoirement. Un vecteur y de taille n est créé. Ce vecteur y stocke une affectation courante.
   Pour $i = 1$ à n, $y(i)$ reçoit aléatoirement la valeur 0 ou 1.
   Pour $k = 1$ à n, la variable $x_{\pi(k)}$ est traitée (les variables sont ensuite parcourues selon l'ordre π) :
   $x_{\pi(k)}$ reçoit temporairement la valeur $y(\pi(k))$.
   Si la variable $x_{\pi(k)}$ apparaît dans une clause unitaire, alors la propagation du littéral unitaire est effectuée.
   Si la variable $x_{\pi(k)}$ apparaît dans deux clauses unitaires contradictoires, alors l'algorithme recommence depuis le début.
   Si aucun de ces deux cas ne se produit, alors la variable $x_{\pi(k)}$ reçoit définitivement la valeur $y(k)$.
   À chaque étape, une vérification est effectuée pour décider si la formule $\mathcal{F}$ est satisfaite par l'affectation donnée par y et si c'est le cas alors l'algorithme s'arrête.
   Ceci est répété jusqu'à ce qu'une affectation satisfaisant la formule $\mathcal{F}$ soit trouvée,

ou jusqu'à une certaine limite de temps fixée.

En 2005, les mêmes auteurs [114] présentent quelques améliorations afin d'optimiser le temps de résolution.

3.2.1.2.2 Méthodes de recherche locale

Dans cette section, nous présentons brièvement quelques algorithmes utilisant la recherche locale pour résoudre des problèmes SAT. Certaines de ces méthodes donnent de bons résultats pour des instances réalisables. Par contre, le principal désavantage de ces méthodes est le fait que, pour les instances non réalisables, elles ne peuvent pas exhiber une preuve de l'incohérence.

Un des premiers algorithmes à employer la recherche locale pour le problème SAT est GSAT (G pour greedy), proposé en 1992 par Selman et al. [127,129]. Le pseudo-code de l'algorithme est présenté dans la Figure 3.4. GSAT exécute une recherche locale gloutonne pour trouver une affectation des variables satisfaisant la formule SAT. Pour chaque variable v on calcule $score(v)$ qui est la différence entre le nombre de clauses satisfaites si la valeur de v est changée ("flippée") et le nombre de clauses satisfaites dans l'affectation actuelle ($score(v)$ peut être positif, nul ou négatif). Selman et al. [129] précisent que si le meilleur score est négatif, alors ils ignorent de tels mouvements et l'algorithme recommence avec une autre affectation (ils ont effectué quelques tests en autorisant de tels mouvements, mais le taux de satisfaction d'instances réalisables est faible). Selman et al. ont comparé le fait d'effectuer un mouvement de score nul ou non. Ils appellent les mouvements qui ont un score nul des mouvements de côté (`sideways move`). Ils ont effectué quelques comparaisons et le fait d'effectuer les mouvements de score nul améliore le taux de réussite donc ces mouvements sont pris en compte dans leur algorithme. Le score de chaque variable est mis à jour à chaque étape de l'algorithme. L'algorithme

commence avec une affectation générée aléatoirement. Ensuite, il change l'affectation de la variable v qui a le plus grand $score(v)$, i.e., qui va engendrer la plus grande augmentation de clauses satisfaites (ces mouvements sont appelés "flips"). Si plusieurs variables ont le même score maximal, alors une variable est choisie au hasard selon une distribution uniforme et la valeur de cette variable est modifiée. De tels changements sont répétés jusqu'à ce qu'un des deux cas suivants soit atteint : soit une affectation satisfaisant la formule est trouvée, soit un nombre maximum de changements (MAX-FLIPS) est atteint. Si c'est le deuxième cas qui est atteint, alors l'algorithme recommence avec une nouvelle affectation générée aléatoirement. Ce processus est répété au plus MAX-ESSAIS fois.

Cet algorithme a contribué au développement des algorithmes incomplets pour SAT basés sur la recherche locale.

```
GSAT
pour i = 1 à MAX-ESSAIS faire
    T ← affectation aléatoire des variables de F;
    pour j = 1 à MAX-FLIPS faire
        si T satisfait F alors
            retourner T;
        sinon
            v ← variable tq score(v) ≥ 0 et score(v) est maximum;
            T ← T avec valeur de v est inversée;
retourner "aucune affectation satisfaisant F n'a été trouvée";
```

Figure 3.4 – Procédure GSAT

GSAT a donné de bons résultats sur différentes instances réalisables.

HSAT [58] est une variante de GSAT où les itérations de recherche locale utilisent des informations basées sur l'historique de la recherche. Quand plusieurs variables ont le même score, HSAT choisit la variable dont la valeur a été changée le moins récemment.

La méthode GSAT avec marche aléatoire proposée par Selman et al. [126, 128] combine la **marche aléatoire** (random walk) avec des algorithmes dérivés de GSAT. À chaque étape, dans le but d'échapper aux minima locaux, la marche aléatoire effectue avec probabilité p un mouvement aléatoire et avec probabilité $1 - p$ un mouvement du type de GSAT. Si un mouvement aléatoire est effectué, une contrainte violée C est choisie au hasard et la valeur d'une variable de C est modifiée afin de satisfaire C.

La méthode WalkSAT est une variante de GSAT avec marche aléatoire, dont les idées ont été premièrement proposées par Selman et al. [128] et qui ont par la suite été précisées par McAllester et al. [103]. La procédure WalkSat est détaillée dans la Figure 3.5.

```
WalkSat
répéter
    C_i ← clause non satisfaite choisie aléatoirement;
    si il existe une variable v dans C_i tq bv(v) = 0 alors
        changer la valeur de v;
    sinon
        avec probabilité p
            v ← une variable de C_i choisie aléatoirement;
            changer la valeur de v;
        avec probabilité (1 − p)
            v ← une variable de C_i avec le plus petit bv(v);
            changer la valeur de v;
jusqu'à ce qu'une solution soit trouvée ;
```

Figure 3.5 – Procédure WalkSat

WalkSAT fonctionne en deux phases. À chaque étape, une première phase consiste à choisir de manière aléatoire une clause C_i non satisfaite. Ensuite, dans la deuxième phase, la valeur d'une des variables de C_i est modifiée pour obtenir une nouvelle affectation. Dans ce cas, la fonction de score est différente de celle utilisée par GSAT avec marche aléatoire. Dans ce cas-ci, nous allons l'appeler $bv(v)$ (elle est appelée *break*

value par McAllester et al. [103]) et $bv(v)$ indique le nombre de clauses qui sont actuellement satisfaites mais qui deviendraient non satisfaites si la valeur de v est changée. En pratique, ce score est une bonne mesure du changement total de nombre de clauses insatisfaites quand une variable v est modifiée. Si dans C_i il existe une variable v telle que $bv(v) = 0$, i.e., le changement de sa valeur permet de satisfaire C_i et n'augmente pas le nombre de clauses non satisfaites, alors cette variable est choisie. Un mouvement de la sorte augmente le nombre de clauses satisfaites d'au moins un. Si aucune variable de la sorte n'existe alors, avec probabilité p, une variable v de C_i est choisie aléatoirement et, avec probabilité $1 - p$, une variable v avec le plus petit $bv(v)$ est choisie et sa valeur est changée. La différence entre GSAT avec marche aléatoire et WalkSAT est subtile. GSAT avec marche aléatoire peut être vue comme ajoutant de la marche aléatoire à un algorithme glouton, tandis que WalkSAT peut être vue comme ajoutant un élément glouton à une heuristique de marche aléatoire.

L'algorithme Novelty présenté par McAllester et al. [103] combinent des algorithmes basés sur l'architecture de WalkSAT avec une méthode de sélection des variables basée sur l'historique des mouvements comme HSAT.

Citons encore la méthode GRASP (*greedy randomized adaptative search procedure*) de Resende et Feo [121] qui est une heuristique randomisée qui fonctionne de façon itérative. Chaque itération de GRASP consiste en deux phases : une étape de construction où une solution admissible est construite itérativement de façon gloutonne et une étape de recherche locale. À chaque itération de l'étape de construction, afin de choisir le prochain élément qui va être ajouté, les candidats sont ordonnés selon une liste en respectant une fonction gloutonne. La méthode est adaptative car le score associé à chaque candidat par cette fonction est mis à jour à chaque étape de la construction selon les changements apportés par le choix de l'élément de l'étape précédente. La méthode est probabiliste car l'élément choisi à chaque étape n'est pas forcément celui en haut de

la liste, mais est choisi au hasard parmi les premiers éléments de la liste. Comme les solutions de cette phase de construction ne sont pas garanties d'être des minima locaux, une recherche locale est ensuite appliquée dans le but d'améliorer cette solution.
Précisons qu'il existe dans la littérature un autre algorithme pour SAT qui s'appelle lui aussi GRASP pour *Generic seaRch Algorithm for the Satisfiability Problem* développé par Marques-Silva et Sakallah [100] et qui n'est pas heuristique. C'est un algorithme complet axé sur l'inévitabilité des conflits durant la recherche. Un de ses points forts est l'augmentation du nombre de retour-arrières grâce à une procédure d'analyse de conflits très efficace.

Mazure et al. [102] proposent une heuristique tabou. Cet algorithme, appelé TSAT, garde une liste tabou de longueur fixe des variables modifiées ("flips") et interdit aux variables de la liste d'être modifiées pendant un certain temps.

Le problème SAT étant très étudié, une multitude d'algorithmes dérivés de ceux présentés ci-dessus existent ainsi que d'autres utilisant par exemple la recherche locale dynamique. Comme le but de cette thèse n'est pas de trouver un nouvel algorithme pour SAT mais de détecter des sous-ensembles incohérents minimaux, les autres méthodes ne vont pas être détaillées. Par contre, nous pouvons référer le lecteur intéressé par d'autres méthodes à lire les articles suivants qui passent en revue les différentes méthodes et qui les évaluent : Gu et al. [70], Gent et Walsh [57] et Hoos et Stützle [76] pour les méthodes utilisant la recherche locale.

3.2.2 Méthodes de résolution des problèmes Max-SAT et Max-SAT pondérés

La plupart des heuristiques développées pour le problème SAT et décrites auparavant peuvent être directement appliquées au problème Max-SAT car elles essaient de déterminer si le problème est réalisable en construisant des affectations qui satisfont le plus de

clauses possibles à chaque étape et qu'elles essaient d'augmenter ce nombre de clauses satisfaites.
L'heuristique GSAT de Selman et al. [129] décrite précédemment a été étendue pour les problèmes Max-SAT pondérés par Jiang et al. [82]. De même, la méthode GRASP de Resende et Feo [121] peut aussi être appliquée aux problèmes Max-SAT et Max-SAT pondéré.

En 1990, Hansen et Jaumard [72] ont proposé pour le problème Max-SAT une méthode similaire à Tabou qu'ils ont appelée *Steepest Ascent Mildest Descent*. Leur méthode a été développée spécialement pour résoudre un problème Max-SAT, mais elle peut aussi résoudre un problème SAT et elle vaut toutes les méthodes de recherche locale résolvant le problème SAT présentées à la section 3.2.1.2.2.

Pour d'autres algorithmes de recherche locale pour le problème Max-SAT, le lecteur peut se référer à l'article de Stützle, Hoos et Roli [131].

La plupart des algorithmes exacts développés pour les problèmes Max-SAT ou Max-SAT pondérés [4, 5, 93, 94] sont basés sur des algorithmes de type "Branch & Bound" en profondeur avec retour-arrière. À chaque noeud de l'arbre de recherche, l'algorithme compare la borne supérieure (UB pour *upper bound*) qui est la valeur de la meilleure solution trouvée jusque-là pour une affectation complète des variables, avec la borne inférieure (LB pour *lower bound*) qui est la somme du nombre de clauses non satisfaites par l'affectation courante plus une sous-estimation du nombre de clauses qui vont devenir non satisfaites si l'affectation partielle courante est complétée. Si $LB \geq UB$ l'algorithme n'évalue pas la branche en dessous du noeud courant et fait un retour-arrière. Si $LB < UB$, l'algorithme essaie de trouver une meilleure solution en augmentant l'affectation partielle courante en affectant une variable de plus, ce qui ajoute deux nouvelles branches à l'arbre courant. La solution de Max-SAT est la valeur que prend UB quand

l'arbre entier a été exploré. La propagation unitaire, présentée précédemment pour les méthodes de type DPLL, [37], pour résoudre le problème SAT ne peut pas être appliquée pour le problème Max-SAT car elle peut augmenter le nombre minimum de clauses non satisfaites (et donc diminuer le nombre maximum de clauses satisfaites). En effet, par exemple la formule $\mathcal{F} = \{x_1\} \wedge \{\bar{x}_1 \vee x_2\} \wedge \{\bar{x}_1 \vee \bar{x}_2\} \wedge \{\bar{x}_1 \vee x_3\} \wedge \{\bar{x}_1 \vee \bar{x}_2\}$ peut satisfaire 4 contraintes au maximum. La première clause est unitaire. Si nous imposons $x_1 = 1$ et que nous propageons cela, nous obtenons la formule : $\mathcal{F}' = \{x_2\} \wedge \{\bar{x}_2\} \wedge \{x_3\} \wedge \{\bar{x}_3\}$. Donc, $\mathcal{F}'$ combinée avec le fait que $x_1 = 1$ est satisfaite obtient un maximum de trois contraintes satisfaites. Quand un branchement est fait sur le littéral l, toutes les clauses contenant l sont supprimées et $\bar{l}$ est supprimé de toutes les clauses le contenant, mais les clauses unitaires obtenues par ces opérations ne sont pas propagées. De même, la règle de la saturation peut augmenter le nombre de clauses non satisfaites. Par contre, la règle du littéral pur ou de la subsomption peuvent être appliquées.

Borchers et Furman [23] présentent un algorithme exact pour les problèmes Max-SAT et Max-SAT pondérés fonctionnant en deux phases. Dans la première phase, ils utilisent l'heuristique GSAT pour trouver une première bonne solution au problème. Ensuite, dans la deuxième phase, ils utilisent une procédure d'énumération basée sur un algorithme de type DPLL pour trouver une solution optimale (prouvable). En fait, la première phase permet d'optimiser la deuxième phase, car elle fournit une borne supérieure sur le nombre minimum de clauses non satisfaites et ceci permet d'élaguer des branches de l'arbre de recherche de la deuxième phase. La valeur de leur borne inférieure est le nombre de clauses non satisfaites par l'affectation partielle courante. De plus, quand la différence entre la borne inférieure et la borne supérieure est exactement un, ils utilisent la règle de propagation unitaire, car dans ce cas-là son utilisation ne peut pas augmenter le nombre de clauses non satisfaites par Max-SAT (comme vu précédemment, cette règle ne peut pas être utilisée dans les autres cas, car son application peut augmenter le nombre de clauses non satisfaites par Max-SAT).

Alsinet et al. [4] présentent un algorithme appelé `Lazy` pour Max-SAT pondéré utilisant le même schéma que Borchers et Furman, mais en ajoutant des raffinements utilisant des prétraitements et des structures de données bien adaptées.

Li et al. [93] présentent cinq façons de calculer la borne inférieure. Comme nous l'avons vu ci-dessus, la borne inférieure, LB, est la somme du nombre de clauses non satisfaites par l'affectation courante plus une sous-estimation du nombre de clauses qui vont devenir non satisfaites si l'affectation partielle courante est complétée. Plus précisément, Li et al. présentent cinq façons d'évaluer cette sous-estimation du nombre de clauses qui vont devenir non satisfaites. Pour cela, ils comptent le nombre de sous-ensembles incohérents disjoints qui peuvent être détectés en utilisant la propagation unitaire.

Li et al. [94] proposent de nouvelles règles de simplification d'une formule Max-SAT ou Max-SAT pondéré. Afin de prouver que ces règles ne modifient pas la formule originale, ils donnent des preuves en utilisant la transformation en programme en nombres entiers d'une formule Max-SAT. Leur algorithme `MaxSatz` utilise ces nouvelles règles de simplification dans le but d'améliorer la qualité des bornes inférieures.

Bonet et al. [22] présentent une nouvelle règle de simplification pour les problèmes Max-SAT et Max-SAT pondérés. Dans le problème Max-SAT, les clauses répétées doivent être gardées. Cette nouvelle règle est appelée *règle de résolution Max-SAT* et contrairement aux règles classiques de résolution, supprime les clauses originales ayant provoqué l'utilisation de la règle. Cette résolution Max-SAT fonctionne de la façon suivante.
Supposons que les deux clauses originales sont : $a \vee x_1 \vee \ldots \vee x_i$ et $\bar{a} \vee y_1 \vee \ldots \vee y_j$.

Alors les clauses obtenues par la règle de résolution Max-SAT sont :

$$
\begin{aligned}
& x_1 \vee \ldots \vee x_i \vee y_1 \vee \ldots \vee y_j \\
& a \vee x_1 \vee \ldots \vee x_i \vee \bar{y}_1 \\
& a \vee x_1 \vee \ldots \vee x_i \vee y_1 \vee \bar{y}_2 \\
& \ldots \\
& a \vee x_1 \vee \ldots \vee x_i \vee y_1 \vee \ldots \vee y_{j-1} \vee \bar{y}_j \\
& \bar{a} \vee y_1 \vee \ldots \vee y_j \vee \bar{x}_1 \\
& \bar{a} \vee y_1 \vee \ldots \vee y_j \vee x_1 \vee \bar{x}_2 \\
& \ldots \\
& \bar{a} \vee y_1 \vee \ldots \vee y_j \vee x_1 \vee \ldots \vee x_{i-1} \vee \bar{x}_j
\end{aligned}
$$

Cette règle est appliquée aux ensembles de clauses dans lesquels une clause peut être répétée. Cette règle préserve le nombre de clauses non satisfaites pour chaque affectation des variables. Bonnet et al. présentent un algorithme complet, qui peut être vu comme une extension pour Max-SAT de l'algorithme de Davis et Putnam [38], utilisant cette nouvelle règle pour Max-SAT. De plus, ils proposent une règle similaire pour la version pondérée.

Citons encore que des algorithmes utilisant la programmation en nombres entiers ont été développés. Par exemple, pour Max-2-SAT, Lewin et al. [91] utilisent un algorithme de plans coupant pour trouver une approximation de la valeur optimale.
De même, Joy et al [84] présentent un algorithme de branchements et coupes (*branch and cut*) qui utilise GSAT comme heuristique primaire au début de l'algorithme. Puis, à chaque noeud de l'arbre de recherche, ils résolvent une relaxation du programme linéaire associé à Max-SAT et ajoutent des coupes.

De même, des algorithmes d'approximation ont été développés, par exemple par Johnson [83] et Dantsin et al. [36].

3.2.3 Recherche de sous-ensembles incohérents minimaux

Nous précisons d'abord les définitions d'IIS de contraintes et de variables dans le cas particulier du problème SAT.

Définition 3.2.5. Étant donné une formule propositionnelle non réalisable $\mathcal{F}$ ayant pour ensemble de clauses $\mathcal{C}$, un *sous-ensemble incohérent irréductible de clauses*, IIS-C, $\mathcal{M} \subseteq \mathcal{C}$, est un sous-ensemble de clauses non réalisable qui devient réalisable dès qu'on lui enlève n'importe quelle clause.

Comme nous l'avons déjà dit, pour le problème SAT, un IIS-C est souvent appelé un MUS (où MUS signifie minimal unsatisfiable subformulaes) ou un MUSC (par exemple par Desrosiers et al. [40]). L'extraction des IIS-C est souvent appelée *MUS selection*.

Clause							$\mathcal{M}_1$	$\mathcal{M}_2$	$\mathcal{M}_3$
C_1	$\neg x_1$	$\vee$	x_2				•		•
C_2	$\neg x_1$	$\vee$	$\neg x_2$				•		•
C_3	x_1	$\vee$	x_3				•	•	•
C_4	x_1	$\vee$	$\neg x_3$				•		
C_5	x_3	$\vee$	$\neg x_5$					•	
C_6	$\neg x_3$	$\vee$	$\neg x_5$					•	•
C_7	$\neg x_3$	$\vee$	x_5					•	•
C_8	$\neg x_1$	$\vee$	x_3	$\vee$	$\neg x_4$			•	
C_9	$\neg x_1$	$\vee$	x_4	$\vee$	x_5			•	

Figure 3.6 – Exemple d'une instance SAT non réalisable.

La figure 3.6 montre une instance CNF comprenant 9 clauses et 5 variables. Ce problème CNF est non réalisable et a trois IIS-C, chacun permettant d'expliquer l'incohérence de cette formule. Par exemple, $\mathcal{M}_1$ exprime l'impossibilité d'affecter des valeurs à x_1, x_2, et x_3 de telle façon à ce que les quatre premières clauses soit simultanément satisfaites.

Il est parfois plus facile de comprendre pourquoi un problème est non réalisable en considérant simplement les variables, plutôt que les contraintes, qui sont impliquées dans l'incohérence. En effet, les problèmes originaux ont généralement moins de variables que de contraintes et les IIS de variables sont généralement plus petits que les IIS de contraintes. Nous allons voir dans le chapitre 4 qu'il est souvent plus facile de trouver d'abord un IIS de variables puis, à partir de ce dernier, il est possible d'en extraire un IIS de contraintes. Étant donné une formule propositionnelle $\mathcal{F}$ agissant sur l'ensemble des variables $\mathcal{X}$, et étant donné un sous-ensemble $\mathcal{Q} \subseteq \mathcal{X}$, la sous-formule de $\mathcal{F}$ induite par $\mathcal{Q}$, notée $\mathcal{F}_{\mathcal{Q}}$, est la formule composée par les clauses C_i dont toutes les variables qui les composent appartiennent à $\mathcal{Q}$. On dit qu'un sous-ensemble Q de variables est réalisable si et seulement si $\mathcal{F}_{\mathcal{Q}}$ est réalisable. Par conséquent, étant donné une affectation partielle dans laquelle seulement les variables de Q obtiennent une valeur, chaque clause C_i qui contient au moins une variable non affectée $x_j \notin Q$ est considérée comme satisfaite. La raison de cela est qu'il est toujours possible de donner une valeur à x_j afin que C_i soit évaluée à *vrai*. Ceci conduit à une variation du problème Max-SAT où des affectations partielles sont autorisées et où le but est de trouver une affectation du plus grand sous-ensemble de variables $\mathcal{Q} \subseteq \mathcal{X}$ possible afin que la sous-formule correspondante $\mathcal{F}_{\mathcal{Q}}$ soit évaluée à *vrai*. De plus, ce problème peut être étendu en donnant à chaque variable $x_i \in \mathcal{X}$ un poids w_i, et en ayant pour but de trouver une affectation qui maximise la somme des poids des variables affectées (ou minimise le poids des variables non affectées).

On peut donner une définition équivalente à la définition précédente en parlant des variables.

Définition 3.2.6. Étant donné une formule propositionnelle non réalisable $\mathcal{F}$ ayant comme ensemble de variables $\mathcal{X}$, un *sous-ensemble incohérent irréductible de variables*, *IIS-V*, $\mathcal{W} \subset \mathcal{X}$, est un sous-ensemble de variables de $\mathcal{X}$, tel que si on retire n'importe quelle variable de ce sous-ensemble $\mathcal{W}$, alors la sous-formule obtenue est réalisable.

Desrosiers et al. [40] appellent les IIS-V pour le problème SAT des MUSV. Si les variables des contraintes composant un IIS-C n'apparaissent pas dans d'autres contraintes du problème, alors elles forment un IIS-V. Par contre, il se peut que l'ensemble des contraintes ne contenant que les variables d'un IIS-V ne constituent pas un IIS-C.
Si nous considérons l'exemple de la Figure 3.6, nous remarquons que celui-ci contient deux IIS-V : $\{x_1, x_2, x_3\}$ et $\{x_1, x_3, x_4, x_5\}$. Le premier IIS-V donne la sous-formule correspondant aux clauses C_1, C_2, C_3 et C_4, qui est aussi la sous-formule correspondant à l'IIS-C $\mathcal{M}_1$. Par contre, le second IIS-V donne la sous-formule correspondant aux clauses C_3, C_4, C_5, C_6, C_7, C_8 et C_9, qui n'est pas minimal dans le sens des IIS-C (i.e., C_4 peut être enlevée sans rendre la sous-formule réalisable).

3.2.3.1 Extraction d'un IIS-C

Contrairement aux méthodes que nous allons présenter au Chapitre 4, qui permettent l'extraction d'IIS-C ou d'IIS-V, les algorithmes qui ont été développés par les autres chercheurs se concentrent uniquement sur la recherche d'IIS-C. Nous allons présenter dans cette section une revue des méthodes existantes pour l'extraction d'un IIS-C.

Bruni [26] propose une méthode adaptative de recherche d'IIS-C. Sa procédure classifie les clauses selon leur "difficulté" qui est définie en analysant l'historique de recherche d'un algorithme complet de type DPLL. En fait, la "difficulté" est proportionnelle au nombre de fois qu'une clause est "visitée" et "échouée" durant la recherche d'un algorithme complet.
Une clause C_j est dite *visitée* si durant l'exploration d'un arbre de recherche (d'un algorithme complet) une affectation d'une variable de C_j est faite dans le but de satisfaire C_j.
Une clause C_j est dite *échouée* si un des deux cas suivants se produit :

- une affectation ayant pour but de satisfaire C_j produit une clause vide ;

– C_j elle-même devient vide suite à une autre affectation.

Soit v_j le nombre de visites à la clause C_j, f_j le nombre d'échecs dus à C_j (i.e., dus aux deux cas décrits ci-dessus), p une valeur (constante) pénalisant les échecs, et l_j la longueur de C_j.

L'évaluation de la difficulté de C_j dans $\mathcal{F}$ est donnée par

$$\phi(C_j) = (v_j + p \cdot f_j)/l_j$$

La procédure de recherche adaptative fonctionne de la façon suivante.

D'abord, il y a une phase de prétraitement de propagation des clauses unitaires. Puis, d branchements sont effectués sur $\mathcal{F}$ (ou moins si $\mathcal{F}$ est résolue avant). Le sous-ensemble initial de clauses, K_0, est vide.

Lors de l'initialisation, un pourcentage fixé c de clauses de $\mathcal{F}$ est ajouté à K_0 pour former K_1, en donnant la priorité aux clauses les plus difficiles. Les clauses restantes forment O_1.

À l'itération k, b branchements sont effectués sur le sous-ensemble K_k (ou moins que b si K_k est résolu avant). Un des trois cas suivants se produit.

1. K_k est non réalisable, donc $\mathcal{F}$ est non réalisable, et K_k est un sous-ensemble non réalisable.
2. Il n'y a pas de résultat après b branchements. Alors, une phase de contraction est effectuée, i.e., un nouveau sous-ensemble, K_{k+1} est formé en sélectionnant un pourcentage c fixé des clauses de K_k, en donnant la priorité aux clauses les plus difficiles. Ensuite, on pose $k := k + 1$ et on recommence.
3. K_k est satisfait par une solution S_k. Alors, une phase d'expansion est effectuée. Un nouveau sous-ensemble K_{k+1} est créé en ajoutant un pourcentage fixé c de clauses de O_k. Ensuite, on pose $k := k + 1$ et on recommence.

Bruni a testé son algorithme sur des instances du Dimacs. Cette procédure fonctionne bien sur des petits exemples, mais très mal sur des grands. De plus, souvent les sous-

ensembles non réalisables obtenus par cette procédure ne sont pas minimaux.

Un *polyèdre* P est un ensemble décrit de la façon suivante : $P = \{x \in \mathbb{R}^n \mid Ax \geq b\}$ où A est une matrice de taille $m \times n$ et b est un vecteur de taille $m \times 1$. Donc, l'ensemble des solutions admissibles d'un programme linéaire est un polyèdre. Définissons un *point entier* comme un point ayant toutes ses composantes entières. Un polyèdre non vide ayant au moins un point intérieur entier vérifie la *propriété de point intérieur entier, (integral-point property)*. Bruni [27] propose une méthode d'extraction d'IIS-C pour des classes de formules CNF ayant la propriété de point intérieur entier. Il utilise pour cela une variante du lemme de Farkas et la résolution d'un programme linéaire. Il présente des résultats encourageants sur des instances provenant de problèmes réels. Par contre, seulement quelques sortes de formules CNF vérifient la propriété de point intérieur entier : parmi elles figurent les instances de Horn (chaque clause a au plus un littéral positif) et renommables de Horn (tous les littéraux peuvent être renommés de façon uniforme de telle sorte que l'instance avec les littéraux renommés soit de type Horn).

L'algorithme AMUSE de Oh et al. [109] est basé sur un algorithme complet de type DPLL. À chaque clause de la formule de départ, une variable supplémentaire est ajoutée. Ensuite, un algorithme complet de recherche est modifié afin de trouver un ensemble non réalisable de clauses (non forcément minimal) grâce aux variables ajoutées. Supposons que la formule CNF originale soit $\mathcal{F} = C_1 \wedge ... \wedge C_m$ où les variables originales sont $\mathcal{X} = \{x_1, ..., x_n\}$. La nouvelle formule obtenue après l'ajout des variables $\mathcal{Y}$ est

$$\mathcal{F}' = (\bar{y}_1 \vee C_1) \wedge ... \wedge (\bar{y}_m \vee C_m)$$

où $\mathcal{Y} = \{y_1, ..., y_m\}$ et $\mathcal{Y} \cap \mathcal{X} = \emptyset$. En posant $y_i = 0$, la clause C_i est désactivée et en posant $y_i = 1$, la clause C_i est activée. Le problème de trouver un IIS-C de $\mathcal{F}$ est réduit à trouver une affectation des variables $\mathcal{Y}$, notée $\mathcal{Y}^*$ telle que $\mathcal{F}'$ avec les variables $\mathcal{Y}^*$

soit non réalisable et $\mathcal{F}'$ avec les variables $\mathcal{Y}'$ est réalisable où les variables de $\mathcal{Y}'$ sont obtenues à partir des variables $\mathcal{Y}^*$ en remplaçant un $y_i = 1$ en $y_i = 0$.

L'algorithme AMUSE utilise un algorithme complet de type DPLL sur $\mathcal{F}'$ pour implicitement rechercher un sous-ensemble non réalisable de clauses de $\mathcal{F}$. Pour cela, AMUSE traite les variables de $\mathcal{X}$ et $\mathcal{Y}$ différemment durant la recherche. Les variables de $\mathcal{X}$ sont traitées de façon normale : elles sont soit affectées par le processus d'affectation soit forcées par le processus de déduction (lors de la propagation des clauses unitaires) et elles sont désaffectées lors des retour-arrières effectués en présence de conflits. Les variables $\mathcal{Y}$ servent à identifier les clauses qui forment le sous-ensemble non réalisable. L'algorithme "force" une variable de $\mathcal{Y}$ à avoir la valeur 1 pour indiquer que la clause correspondante de $\mathcal{F}$ est candidate de l'IS. Afin de trouver cet IS, l'algorithme stocke les variables dans différents ensembles : $\mathcal{X}_{affecté}$ est formé par l'ensemble des variables de $\mathcal{X}$ qui ont été affectées par le processus d'affectation ou de déduction et $\mathcal{Y}_{forcé}$ est formé par l'ensemble de variables de $\mathcal{Y}$ qui ont été forcées à prendre la valeur 1 (nous verrons ci-dessous quand ceci a lieu). Au départ, les ensembles $\mathcal{X}_{affecté}$ et $\mathcal{Y}_{forcé}$ sont vides, et ils sont graduellement étendus. Comme la formule $\mathcal{F}$ n'est pas réalisable, à un certain moment de l'exécution de la recherche arborescente sur $\mathcal{F}'$, il va y avoir une contrainte C_i dont toutes les variables appartenant à $\mathcal{X}$ sont non satisfaites. Ceci va forcer y_i à prendre la valeur 0 pour que $\mathcal{F}'$ soit satisfaite. Ceci indique que la clause C_i doit être désactivée afin de trouver une solution pour $\mathcal{F}'$ et que l'algorithme puisse identifier C_i comme clause candidate à l'IS en construction. Alors, une des variables de $\mathcal{X}_{affecté}$ ayant provoqué cette situation (i.e., ayant impliqué $y_i = 0$) est désaffectée et y_i est forcé à prendre la valeur 1.

La recherche se poursuit jusqu'à ce que la combinaison de $\mathcal{X}_{affecté}$ et $\mathcal{Y}_{forcé}$ provoque la création d'une clause apprise ω. Si la formule de départ $\mathcal{F}$ est bien non réalisable, alors obligatoirement une des clauses apprises obtenues lors de l'exécution de l'algorithme va contenir uniquement des variables de $\mathcal{Y}$: $\omega_j = \bar{y}_{j_1} \vee \cdots \vee \bar{y}_{j_k}$. Ceci se produit quand l'affectation actuelle des variables $y_{j_1}, \cdots, y_{j_{k-1}}$ force une variable y_{j_k} à prendre la valeur

0 (i.e., aucune variable de $\mathcal{X}$ n'est impliqué dans le processus ayant forcé y_{j_k} à prendre la valeur 0). Cette clause ω_j identifie implicitement un sous-ensemble non réalisable de clauses de taille k formé par les clauses $C_{j_1}, \cdots, C_{j_k}$.
Les différentes décisions faites sur les variables de $\mathcal{X}$ lors de la recherche arborescente permettent d'obtenir des IS de clauses différents. Sur certaines instances les IS de clauses obtenus sont des IIS de clauses. Les auteurs présentent des résultats de leur algorithme sur des instances provenant de problèmes réels (DaimlerChrysler benchmarks). Cet algorithme fonctionne bien sur de petites instances mais est moins performant sur de grandes instances.

Zhang et Malik [140] proposent une méthode appelée `zCore` pour extraire des sous-ensembles non réalisables minimaux basée sur l'apprentissage lors de l'exécution d'un algorithme complet. Cette méthode travaille sur le graphe de résolution qui peut être généré lors de la résolution d'une formule SAT. Un exemple de graphe de résolution est présenté dans la Figure 3.7.

Le graphe de résolution est un graphe orienté sans cycle. Chaque sommet du graphe est une clause. Les sommets sans prédécesseur sont les clauses du problème de départ. Les noeuds intérieurs sont les clauses générées dans le processus de résolution (dites clauses apprises). Les arcs représentent la résolution. Le noeud final (sans successeur) représente la clause vide obtenue lors de la résolution permettant de dire que la formule est non réalisable. Les sommets (clauses) initiaux à partir desquels un chemin jusqu'au sommet final (clause vide) existe forment le sous-ensemble de clauses qui forme un sous-ensemble incohérent de contraintes. Les clauses racines qui ne sont pas des ancêtres de la clause vide ne sont pas nécessaires à la preuve de non réalisabilité. Par exemple, dans la Figure 3.7, les clauses C_3, C_4, C_6, C_7 et C_8 forment un sous-ensemble incohérent de contraintes. Ce processus peut être répété et appliqué au sous-ensemble incohérent de contraintes trouvé précédemment afin d'extraire un nouveau sous-ensemble incohérent de contraintes de taille plus petite. Les sous-ensembles de contraintes obtenus sont inco-

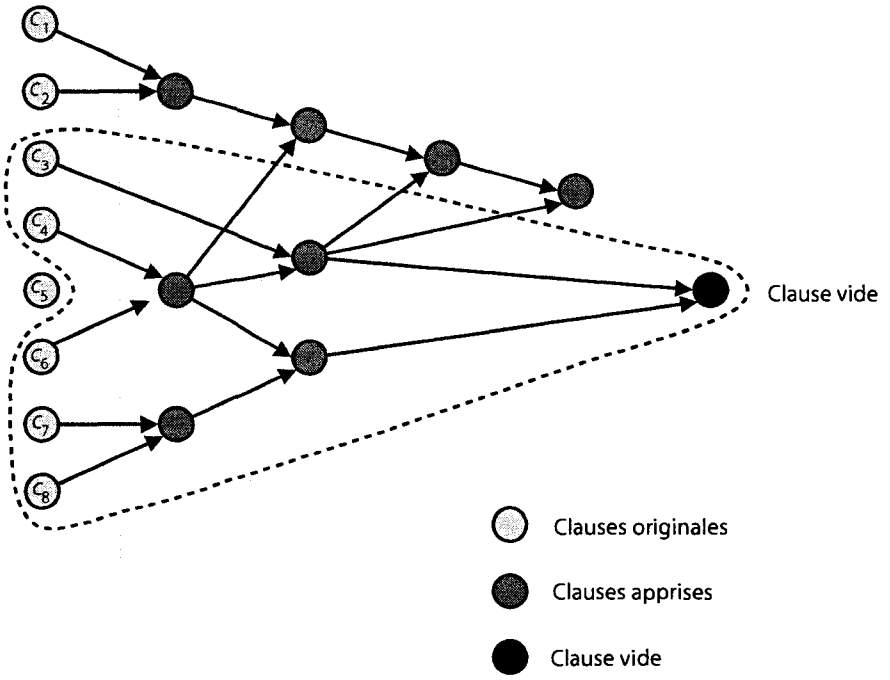


Figure 3.7 – Un exemple de graphe de résolution.

hérents mais pas forcément minimaux.

Cet algorithme est développé pour des grandes instances provenant d'applications réelles. Un des désavantages d'une telle procédure est qu'elle peut échouer sur des grandes instances à cause d'un dépassement de mémoire.

Zhang et Malik présentent des tests sur des instances provenant de problèmes réels. Les résultats montrent que leur algorithme fonctionne bien pour extraire des petits IIS-C.

Gershman et al. [59] ont repris les idées de Zhang et Malik afin de créer un algorithme, appelé `Trimmer`, pour trouver des petits sous-ensembles incohérents (pas forcément minimaux). `Trimmer` parcourt le graphe de résolution et détermine quels noeuds du graphe domine d'autres noeuds, afin d'éliminer rapidement le parcours de ces autres noeuds.

Mazure et al. [101] proposent une méthode de recherche de sous-ensembles incohérents minimaux basée sur un algorithme du type de TSAT. Leur algorithme est basé sur

l'hypothèse que les clauses les plus souvent non satisfaites durant l'exécution d'une recherche locale devraient appartenir aux IIS-C de la formule. Quand TSAT, ou un autre algorithme de type GSAT, est utilisé sur une instance SAT non réalisable, des compteurs leur permettent de séparer le problème en deux parties : une réalisable et une non réalisable. Plus précisément, quand TSAT est utilisé sur une instance SAT, ils utilisent des compteurs sur les instances pour lesquelles l'algorithme échoue à montrer que l'instance est réalisable. Une trace de TSAT est enregistrée : pour chaque clause, en prenant chaque changement ("flip") comme pas de temps, le nombre de fois durant lesquelles cette clause est non satisfaite est mis à jour. Une trace similaire est enregistrée pour chaque littéral en comptant le nombre de fois qu'il apparaît dans des clauses non satisfaites. Ils avancent l'hypothèse que les clauses les plus souvent non satisfaites devraient appartenir à un IIS-C de l'instance SAT. De même, ils avancent l'hypothèse que les littéraux ayant les plus grands scores devraient faire partie de l'IIS-C.

Ces idées sont reprises par Grégoire et al. [63–66] et modifiées de la façon suivante. La version de Mazure et al. [101] augmente le compteur des contraintes falsifiées lors de chaque flip même si celles-ci ne font pas partie d'un IIS-C. Pour essayer de contrer cela, il faut tenir compte du voisinage des clauses falsifiées et n'augmenter le score d'une clause non satisfaite que si le fait de la rendre satisfaite aboutirait à la non satisfaction d'autres clauses. Ils définissent les notions de clause *unisatisfaite* et *critique*. Une clause C est *unisatisfaite* par une affectation des variables si et seulement si exactement un littéral de C est satisfait. Une clause C non satisfaite par rapport à une affectation des variables est *critique* par rapport à cette affectation si et seulement si l'opposé de chacun des littéraux de C est le seul littéral satisfait d'au moins une clause unisatisfaite. Par exemple, supposons que l'ensemble des clauses d'une formule non satisfaite est $\mathcal{C} = \{C_1 = \{x_1 \vee x_2 \vee x_3\}, C_2 = \{x_1 \vee \bar{x}_2\}, C_3 = \{x_2 \vee \bar{x}_3\}, C_4 = \{\bar{x}_1 \vee x_3\}, C_5 = \{\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3\}\}$ et supposons que nous ayons l'affectation suivante des variables $x_1 = 1, x_2 = 1, x_3 = 1$. Alors, les clauses unisatisfaites sont C_2, C_3 et C_4 et

la contrainte C_5 est critique. Le but de leur méthode est alors d'augmenter uniquement les compteurs des clauses critiques. De plus, le score des celles-ci est pondéré par le nombre de clauses unisatisfaites qui leur sont liées (afin de tenir compte de la longueur des clauses critiques). L'idée générale de leur algorithme *AOMUS* (*Approximate One MUS*) est de partir d'une formule non réalisable. Ensuite, tant que la recherche locale n'arrive pas à trouver une affectation des variables satisfaisant toutes les variables de la formule actuelle, les clauses avec les plus petits scores sont supprimées de la formule. À chaque étape, l'ensemble des clauses constituant la formule actuelle est sauvegardé. La dernière étape pour laquelle la recherche locale n'arrive pas à satisfaire la formule est vérifiée par une méthode exacte. Si l'incohérence est vérifiée alors on a un IS de clauses. Ensuite à partir de cet IS de clauses, on peut définir exactement un IIS-C en le minimisant. Ceci se fait en supprimant successivement chacune des clauses de l'approximation et en testant si la sous-formule obtenue est toujours non réalisable. Si tel est le cas, on peut effectivement faire cette suppression.

De récents travaux théoriques ont montré que le fait de décider si une formule CNF contient un IIS-C de déficience fixée k (la déficience k est la différence entre le nombre de clauses et le nombre de variables), pour tout $k \in \mathbb{N}$, est NP-complet, mais des algorithmes efficaces ont pu être développés pour des petites valeurs de k [28, 43].

3.2.3.2 Méthodes de minimisation

La plupart des méthodes décrites ci-dessus permettent de trouver des sous-formules non réalisables, mais n'offrent pas de garantie de minimalité (au sens de l'inclusion). En effet, vérifier l'incohérence minimale est un problème difficile connu pour être D^P-complet (montré par Papadimitriu et Wolfe [111]). La classe D^P est définie par tous les "langages" qui peuvent être considérés comme la différence de deux langages dans NP ou de façon équivalente qui sont l'intersection d'un langage dans NP et d'un dans

$CoNP$ (Papadimitriu et Wolfe [111] et Papadimitriu et Yannakakis [112]). Un problème D^P-complet est équivalent à résoudre un problème SAT - NON SAT défini de la façon suivante : étant donné deux formules propositionnelles $\mathcal{F}_1$ et $\mathcal{F}_2$, est-il vrai que $\mathcal{F}_1$ est réalisable et $\mathcal{F}_2$ est non réalisable ? Les problèmes D^P-complets sont à la fois NP-difficiles et $CoNP$-difficiles.

Du fait que certaines applications demandent la minimalité des sous-formules incohérentes, des algorithmes ont été développés afin de minimiser les sous-ensembles incohérents de clauses trouvés.

L'algorithme `MUP` de Huang [78] permet de prouver l'incohérence minimale de certaines instances et permet d'extraire un IIS-C à partir d'une sous-formule non réalisable (non forcément minimale). Il est particulièrement intéressant d'utiliser MUP après avoir utilisé un algorithme rapide d'extraction d'une sous-formule non réalisable (comme par exemple zCore ou AMUSE) et de vérifier la minimalité de celui-ci ou alors de le minimiser avec `MUP`.

L'algorithme de Huang se base sur deux principes. Premièrement, il réduit le problème de définir si une formule est incohérente minimale en un problème de comptage de modèles (nombre d'affectations des variables telles que la formule est évaluée à vraie) d'une formule CNF augmentée. Il effectue le comptage de modèles ainsi que l'élimination des clauses superflues à l'aide de diagrammes binaires de décision (BDD).

Comme pour `AMUSE`, Huang ajoute des nouvelles variables à sa formule originale, par contre il en ajoute moins que le nombre de clauses. Soit $\mathcal{F} = C_1 \wedge ... \wedge C_m$ la formule qu'on veut vérifier. Soit $\mathcal{F}_i$ la formule obtenue en enlevant la clause C_i. Le but est de montrer la non satisfaisabilité de $\mathcal{F}$ et la satisfaisabilité de tous les $\mathcal{F}_i$. Il y a donc $m+1$ formules à tester.

Huang introduit $k = \lceil log(m+1) \rceil$ nouvelles variables $Y = \{y_1, ..., y_k\}$. À partir de ces k nouvelles variables il construit des nouveaux termes qu'il appelle "*minterms*". Ceux-ci sont construits de la façon suivante : c'est une conjonction de k littéraux où chaque variable apparaît exactement une fois. Il y a 2^k minterms. Par exemple, si $k = 2$, les 4

minterms sont $y_1 \wedge y_2$, $\bar{y}_1 \wedge y_2$, $y_1 \wedge \bar{y}_2$, $\bar{y}_1 \wedge \bar{y}_2$. Ensuite, les m différents minterms, notés M_i, sont ajoutés à $\mathcal{F}$ afin de former $\mathcal{F}' = \bigwedge_{i=1}^{m}(M_i \vee C_i)$. Pour chaque affectation des variables Y, il y a un unique M_i, $1 \leq i \leq 2^k$, qui est évalué à vrai, les autres sont évalués à faux. L'évaluation de M_i à vrai permet de désactiver la clause C_i et son affectation à faux permet au contraire d'activer C_i. La formule est non réalisable minimale si et seulement si $\mathcal{F}'$ possède exactement m modèles sur les variables de Y. Pour pouvoir compter les modèles et donc décider si la formule est minimale non réalisable, Huang utilise des diagrammes de décision binaires (BDD) sur les variables Y. Ceci permet aussi en cas de non minimalité de déterminer les clauses à enlever en répétant le processus d'utilisation des BDD.

MUP est surtout efficace pour prouver la minimalité d'une sous-formule incohérente ou pour minimiser une sous-formule incohérente trouvée par un autre algorithme (`zCore` ou `AMUSE` par exemple), mais n'est pas idéal pour chercher un IIS-C à partir de la formule de départ.

Zhang et Malik [140] proposent en même temps que leur algorithme `zCore` un algorithme de minimisation appelé `zMinimal`. Celui-ci enlève successivement toutes les clauses de la formule et teste avec `zChaff` si la sous-formule obtenue est toujours incohérente ou non. Si tel est le cas, alors la clause est définitivement supprimée et l'algorithme continue avec la clause suivante. Cet algorithme est très similaire à l'algorithme `Removal` présenté pour résoudre le LSCP par Galinier et Hertz [48] et que nous allons présenter à la section 4.1.2.

3.2.3.3 Extraction de tous les IIS-C d'un problème

Pour restaurer la satisfaisabilité d'une formule SAT il faut réparer tous ses IIS-C. C'est pour cela que de certains chercheurs se sont intéressés à l'extraction de tous les IIS-C d'un problème.

Comme nous l'avons vu dans la Section 3.1, Bailey et Struckey [11] ont proposé une adaptation de la méthode DAA pour trouver tous les MSS d'un problème.

Liffiton et Sakallah [95] proposent un algorithme similaire à celui de Bailey et Stuckey dans le sens qu'eux aussi utilisent la dualité entre les MSS et les IIS-C. Leur algorithme CAMUS fonctionne en deux phases, et le but de cet algorithme est d'obtenir tous les IIS-C. Pendant la première phase, l'algorithme construit l'ensemble G de tous les coMSS. Pendant la deuxième phase, il construit tous les hitting sets minimaux de G, i.e., tous les IIS-C. Dans la première phase, pour trouver tous les coMSS, ils doivent trouver tous les MSS. Ceux-ci sont trouvés de façon incrémentale en résolvant des problèmes Max-SAT. Pour cela, ils considèrent le problème où chaque clause est augmentée du littéral négatif d'une nouvelle variable (comme pour AMUSE) afin de pouvoir activer ou désactiver la clause. Ils utilisent une borne qui fixe le nombre maximal de nouvelles variables pouvant être fixées à faux (donc le nombre de clauses pouvant être désactivées). Cette borne vaut 1 au départ et augmente à chaque itération. À la deuxième phase, ils proposent une méthode pour extraire un IIS en temps polynomial. Pour cela, à chaque itération, un coMSS est choisi parmi l'ensemble des coMSS et une clause de ce coMSS est choisie. La clause est ajoutée à l'IIS en construction. Ensuite, toutes les autres clauses du coMSS choisi sont supprimées du problème restant. Puis chacun des coMSS contenant la clause choisie est supprimé. Quand il ne reste plus de coMSS, l'algorithme s'arrête. Ensuite, en combinant la technique pour extraire un IIS et un branchement sur les deux décisions que sont le choix du coMSS et de la clause, ils parviennent à extraire tous les IIS. Cet algorithme donne de meilleurs résultats que celui de Bailey et Stuckey.

Dans un autre article, Liffiton et Sakallah [96] présentent plus en détail les bases théoriques sur lesquelles est basée leur méthode et proposent certaines variantes pour extraire des résultats partiels quand l'extraction de tous les IIS-C prend trop de temps. Les coMSS sont ici appelés des ensembles de correction minimaux, MCS (pour *minimal*

correction subset).

Grégoire et al., [67] et [69], combinent la méthode de Liffiton et Sakallah [95] pour trouver tous les IIS-C à un prétraitement utilisant la recherche locale. Celle-ci joue le rôle d'oracle pour trouver des coMSS potentiels. En fait, c'est uniquement la première phase de l'algorithme de Liffiton et Sakallah qu'ils modifient. La méthode de Grégoire et al. s'appelle HYCAM pour HYbridization for Computing All Muses. Leur méthode est basée sur la notion des clauses critiques (vues à la page 77) et sur la propriété suivante. Soit une formule propositionnelle $\mathcal{F}$ et une affectation I de $\mathcal{F}$ et soit $\mathcal{C}'$ un sous-ensemble des contraintes de $\mathcal{F}$ tel que toutes les clauses de $\mathcal{C}'$ sont falsifiées par I, alors $\mathcal{C}'$ ne peut pas être un coMSS quand il existe au moins une clause de $\mathcal{C}'$ qui n'est pas critique par rapport à I. Cette recherche locale pour trouver les coMMS est plus efficace en termes de temps de calcul que la méthode de Liffiton et Sakallah. Par contre, elle n'est pas garantie d'être complète, car elle peut manquer des coMSS et certains des candidats identifiés peuvent ne pas être des coMSS.

3.2.3.4 Extraction de l'IIS-C de cardinalité minimum

Certains auteurs se sont intéressés à l'extraction de l'IIS-C de cardinalité minimum (appelés parfois *MCUS* pour le terme anglais *minimum cardinality unsatisfiable subformulas*).

C'est le cas de Lynce et Marques-Silva [98]. Comme pour l'algorithme AMUSE, ils ajoutent à la formule de départ m nouvelles variables $\mathcal{Y}$. Les contraintes sont modifiées de la façon suivante. Il y a une nouvelle variable y_i par contrainte C_i qui permet ou non de sélectionner la clause. Les nouvelles clauses C'_i sont construites de la façon suivante : $C'_i = \bar{y}_i \wedge C_i$. Appelons la nouvelle formule $\mathcal{F}'$. Si toutes les variables y_i obtiennent la valeur 0, alors $\mathcal{F}'$ est réalisable. Pour chaque affectation des variables y_i aboutissant à une formule non réalisable, le nombre de variables de $\mathcal{Y}$ qui ont la va-

leur 1 indique le nombre de clauses que contient le sous-ensemble non réalisable. Donc le sous-ensemble non réalisable de taille minimum est obtenu par l'affectation des variables y_i ayant le plus petit nombre de variables avec la valeur 1 et tel que $\mathcal{F}'$ est non réalisable. L'idée de leur algorithme pour trouver un IIS-C de cardinalité minimum est d'utiliser un algorithme de type DLL connu, zChaff de Zhang et Malik [140], sur le nouveau problème. Les variables sont organisées en deux ensembles disjoints $\mathcal{X}$ (variables originales) et $\mathcal{Y}$ (nouvelles variables). Les décisions sont d'abord faites sur les variables $\mathcal{Y}$ et après sur les variables $\mathcal{X}$. Chaque fois que la recherche effectue un retour-arrière en passant d'un niveau d'une variable de $\mathcal{X}$ à un niveau d'une variable de $\mathcal{Y}$, alors un sous-ensemble non réalisable de clauses est trouvé. Cet IIS-C est formé par les clauses C_i pour lesquelles $y_i = 1$. Après que toutes les affectations des variables $\mathcal{Y}$ aient été évaluées, le sous-ensemble non réalisable avec le plus petit nombre de clauses est l'IIS-C de cardinalité minimum recherché. Cette méthode de recherche teste donc toutes les combinaisons de contraintes afin de trouver l'IIS de cardinalité minimum. Comme l'espace de recherche est très grand (taille 2^{n+m}), ils proposent des améliorations afin de le réduire. La première consiste à trouver un premier sous-ensemble non réalisable avec zChaff [140] et ensuite à utiliser la taille de celui-ci comme borne supérieure sur la taille du sous-ensemble minimal lors de la recherche en ajoutant une contrainte de cardinalité au problème. De même, chaque nouveau sous-ensemble trouvé en cours d'exécution peut améliorer cette borne. Les autres améliorations consistent à utiliser les clauses apprises lors de l'exécution du branch-and-bound afin de réduire l'espace de recherche et d'éviter de trouver deux fois la même solution. Comme le nombre de sous-formules non réalisables peut être exponentiel par rapport au nombre de ses variables, même avec les améliorations proposées, la taille des problèmes qui peuvent être résolus par cette méthode est très petite.

Mneihmeh et al. [106] présentent un algorithme de type Branch-And-Bound qui utilise les solutions du problème Max-SAT de façon itérative afin de générer des bornes infé-

rieures et supérieures sur la taille de l'IIS-C de cardinalité minimum et afin de vérifier les sous-ensembles obtenus et de pouvoir brancher sur des sous-formules particulières afin de trouver l'IIS-C de cardinalité minimum.

Précisons aussi que les méthodes essayant d'extraire tous les IIS-C d'une formule propositionnelle (vues précédemment à la Section 3.2.3.3), donnent l'IIS-C de cardinalité minimum dans le cas où elles ont réussi à extraire tous les IIS-C.

L'algorithme `HittingSet` de Galinier et Hertz [48] appliqué à un CSP permet d'extraire un IIS de cardinalité minimum. Il est donc applicable au problème SAT. Comme nous allons utiliser cet algorithme pour extraire des IIS de variables et de contraintes de cardinalité minimum, nous le présenterons en détail à la section 4.1.4.

3.3 Conclusion

Dans ce chapitre, nous avons présenté une revue de la littérature des méthodes de recherche d'IIS dans des CSP généraux et pour le problème de k-coloration de graphe. Puis, nous avons présenté des méthodes de résolution du problème SAT ainsi qu'une revue de la littérature des méthodes d'extraction d'IIS pour le problème SAT.

Dans le chapitre qui suit (chapitre 4), nous présentons des algorithmes de détection automatique d'IIS de contraintes ou de variables pour le problème SAT.

CHAPITRE 4

UTILISATION D'HEURISTIQUES POUR TROUVER DES SOUS-ENSEMBLES INCOHÉRENTS MINIMAUX POUR LE PROBLÈME SAT

Dans ce chapitre nous présentons des algorithmes originaux permettant d'obtenir des IIS de contraintes et de variables pour le problème SAT. Nous montrons que ces algorithmes peuvent être appliqués à de relativement grandes instances, grâce à l'utilisation d'heuristiques. Nous présentons aussi un algorithme permettant d'obtenir des IIS de cardinalité minimum. Puis, nous donnons des heuristiques qui aident à trouver des IIS plus petits ou plus denses, ceux-ci étant plus utiles pour diagnostiquer des systèmes incohérents, ainsi qu'une heuristique permettant d'accélérer la recherche. Puis, nous présentons des résultats expérimentaux et nous comparons ces résultats avec ceux de [26, 78, 95, 101, 106, 109, 140].
Les résultats de ce chapitre sont présentés dans [40], qui a été accepté pour publication dans *Journal of Combinatorial Optimization.*

4.1 Algorithmes de détection d'IIS

Les algorithmes présentés dans ce chapitre sont basés sur les méthodes proposées par Galinier et Hertz [48] pour le problème de recouvrement de grands ensembles (*large set covering problem, LSCP*). Leur article démontre que le problème consistant à trouver des IIS dans des problèmes de satisfaction de contraintes non réalisables peut être formulé comme un problème de recouvrement de grands ensembles (LSCP). Or, comme nous l'avons vu dans le chapitre 2, le problème SAT est un problème de satisfaction de contraintes. Donc, les algorithmes de Galinier et Hertz [48] peuvent être appliqués au problème SAT. Ces algorithmes ont auparavant été appliqués avec succès pour trouver

des sous-graphes critiques du point de vue des arêtes et des sommets et résoudre le problème de coloration de graphe (Desrosiers et al. [39]). De plus, comme les propriétés données dans l'article de Galinier et Hertz [48] sont valides pour tout CSP, nous nous référerons à cet article pour les preuves.

Définition 4.1.1. Soit E un ensemble de n éléments. Soit $E_1, \ldots, E_m$ m sous-ensembles de E dont l'union vaut E (i.e., $\bigcup_{i=1}^{m} E_i = E$). Un sous-ensemble I de $\{1, \ldots, m\}$ tel que $\bigcup_{i \in I} E_i = E$ est appelé *un recouvrement* de E. *Le problème de recouvrement à coût unitaire (*de l'anglais *unicost set covering problem, USCP)* consiste à déterminer un recouvrement I de E de cardinalité minimum.

Définition 4.1.2. Soit E un ensemble de n éléments. Soit $E_1, \ldots, E_m$ m sous-ensembles de E dont l'union vaut E. E et les sous-ensembles E_i ne sont pas donnés en extension car ils peuvent être très grands. Soit une fonction φ telle que $\varphi(e,i) = \begin{cases} 1 & \text{si } e \in E_i \\ 0 & \text{sinon} \end{cases}$. De plus, soit une fonction ω qui associe un poids $\omega(i)$ à chaque ensemble E_i et une procédure $MinW(\omega)$ qui retourne un élément $e \in E$ tel que $\sum_{e \in E_i} \omega(i)$ est minimum. Le *problème de recouvrement de grands ensembles (*de l'anglais *Large Set Covering Problem, LSCP)* consiste à déterminer grâce à φ et $MinW$ un recouvrement minimal de E (i.e., au sens de l'inclusion si on supprime n'importe quel ensemble de $MinW$, alors les ensembles restants ne forment plus un recouvrement de E). De plus, le problème *minimum LSCP* consiste à déterminer avec φ et $MinW$ un recouvrement minimum de E (i.e., de cardinalité minimum).

La recherche d'IIS de contraintes ou de variables dans des CSP non réalisables sont des cas particuliers du LSCP. En effet, supposons que nous ayons un CSP non réalisable. Si nous voulons faire la recherche d'un IIS de contraintes, nous définissons les élé-

ments de E comme n'importe quelle affectation complète des variables de $\mathcal{X}$. À chaque contrainte C_i de $\mathcal{C}$, nous associons le sous-ensemble E_i de E contenant toutes les affectations qui violent C_i. Un recouvrement minimal de E est donc un IIS de contraintes. Alors, la fonction φ est la suivante : $\varphi(e,i) = \begin{cases} 1 & \text{si l'affectation complète } e \text{ viole } C_i \\ 0 & \text{sinon} \end{cases}$.
La fonction ω pondère les contraintes du CSP et la procédure $MinW(\omega)$ retourne une affectation complète e qui minimise la somme des poids des contraintes violées par e.
Dans le cas de la recherche d'un IIS de variables d'un CSP non réalisable, nous définissons un élément de E comme n'importe quelle affectation partielle légale. À chaque variable x_i de $\mathcal{X}$ est associé le sous-ensemble E_i de E qui contient toutes les affectations partielles légales dans lesquelles x_i n'est pas affectée. À nouveau, la résolution du problème LSCP permet de trouver un IIS de variables. Pour cela, la procédure φ est définie de la façon suivante : $\varphi(e,i) = \begin{cases} 1 & \text{si } x_i \text{ n'est pas affectée dans l'affectation} \\ & \text{partielle } e, \\ 0 & \text{sinon} \end{cases}$.
Aussi, ω pondère les variables de $\mathcal{X}$ et $MinW$ retourne une affectation partielle légale e qui minimise la somme des poids des variables non affectées dans e.

La procédure $MinW$ résout principalement des problèmes NP-difficiles, que ce soit dans la recherche d'IIS de variables ou de contraintes. C'est pour cette raison, que suivant le type de CSP auquel nous sommes confrontés il pourra être utile de remplacer cette procédure par une version heuristique.

Dans la section suivante, nous allons présenter les trois algorithmes proposés par Galinier et Hertz [48], i.e., `Removal`, `Insertion` et `HittingSet` de façon générale (exacte et heuristique) et nous allons préciser comment nous les avons adaptés au problème SAT.

4.1.1 Présentation générale des algorithmes d'extraction de sous-problèmes incohérents minimaux

Pour détecter des sous-ensembles incohérents minimaux de contraintes ou de variables, trois types d'approches sont étudiées par Galinier et Hertz [48].

1. **Par suppression :** On part d'un ensemble non réalisable de contraintes (ou variables) et l'on supprime des contraintes (ou variables) tant que l'ensemble reste non réalisable.
2. **Par insertion :** On part de l'ensemble vide et on ajoute des contraintes (variables), une à une, jusqu'à ce que l'ensemble de contraintes (variables) devienne non réalisable.
3. **Par recouvrement :** On détermine des sous-ensembles de contraintes (variables) tels que si on supprime n'importe lequel de ces sous-ensembles, le système devient réalisable. Ensuite, on détermine un recouvrement minimal de ces sous-ensembles.

Afin de présenter une seule version des algorithmes de détection d'IIS, nous allons introduire des notations génériques qui vont avoir une signification différente selon que nous cherchons un IIS de variables ou de contraintes.

Ainsi, si notre but est de trouver des IIS de contraintes, notons S l'ensemble des contraintes C d'un CSP non réalisable et S' un sous-ensemble de S. Comme nous l'avons vu précédemment, e est une affectation complète des variables. Notons $f_S(e)$ le nombre de contraintes de S violées par l'affectation e. Soit S_1 et S_2 deux sous-ensembles disjoints de contraintes. Alors $MIN(S_1, S_2)$ est une procédure qui produit une affectation qui minimise la fonction $\alpha \cdot f_{S_1}(e) + f_{S_2}(e)$ où α est plus grand que $|S_2|$. Donc cela veut dire que $MIN(S_1, S_2)$ détermine une affectation complète des variables qui satisfait le plus de contraintes possibles de S_1 et, parmi toutes ces affectations possibles, produit celle qui viole le moins de contraintes de S_2. Les contraintes de S_1 sont considérées comme dures

et celles de S_2 comme molles. Alors, S' est non réalisable si et seulement si n'importe quelle affectation e obtenue comme valeur de sortie de $MIN(S', \emptyset)$ ou $MIN(\emptyset, S')$ est telle que $f_{S'}(e) > 0$.

Par contre, si notre objectif est de trouver des IIS de variables, S est l'ensemble des variables $\mathcal{X}$ d'un CSP non réalisable, e est une affectation partielle légale des variables de S et S' est un sous-ensemble de variables de S. Dans ce cas-ci, $f_S(e)$ est le nombre de variables non affectées dans e. Soit S_1 et S_2 deux sous-ensembles de variables de S. Alors $MIN(S_1, S_2)$ est une procédure qui produit une affectation partielle optimale qui minimise la fonction $\alpha \cdot f_{S_1}(e) + f_{S_2}(e)$, où α est un nombre plus grand que $|S_2|$. Donc, en fait, $MIN(S_1, S_2)$ détermine une affectation partielle légale qui affecte le plus de variables possible de S_1 et parmi toutes ces affectations possibles retourne celle qui contient le moins de variables de S_2 non affectées. Le sous-ensemble S' est non réalisable si et seulement si $MIN(S', \emptyset)$ ou $MIN(\emptyset, S')$ est tel que $f_{S'}(e) > 0$.

Dans le cas particulier du problème SAT, MIN correspond à une procédure résolvant le problème Max-SAT pondéré (que nous allons abréger dans la suite du texte `MaxWSAT`) et où toutes les contraintes ou les variables de S ont un poids associé.
Nous utilisons les notations suivantes. $F_{S_i}(e_i)$ est formé par l'ensemble des contraintes de S_i violées par l'affectation e_i si nous effectuons une recherche d'IIS de contraintes et est formé par l'ensemble des variables de S_i non affectées dans l'affectation e_i si nous effectuons une recherche d'IIS de variables. $f_{s_i}(e_i)$ est le nombre de contraintes de S_i violées par e_i ou de variables de S_i non affectées dans e_i.
Nous allons aussi présenter des versions heuristiques des algorithmes de recherche d'IIS. Dans ces cas-là, une version heuristique de MIN est utilisée et elle est appelée $HMIN$. Dans le cadre du problème SAT, $HMIN$ est une heuristique résolvant le problème MaxWSAT.

4.1.2 L'algorithme `Removal`

L'algorithme `Removal` est peut-être le plus simple de tous les algorithmes de recherche d'IIS. En effet, étant donné un problème non réalisable, l'algorithme `Removal` procède avec une approche "de haut en bas", supprimant les contraintes ou les variables une par une, et réintroduisant celles qui rendent le problème réalisable. Comme déjà mentionné dans le Chapitre 3 (revue de la littérature) des approches similaires ont déjà été proposées, comme par exemple l'algorithme `zMminimal` de Zhang et Malik [140] pour le problème SAT, l'algorithme de suppression de Chinneck [31] pour la programmation linéaire, ou l'algorithme de Herrmann et Hertz [74] pour la détection de sous-graphes sommets-critiques pour le problème de k-coloration de graphes.

Dans la Figure 4.1 est présenté l'algorithme `Removal` exact. Il détermine un IIS en $|S|$ étapes.

`Removal`

Entrée : un ensemble non réalisable S.
Sortie : un IIS S'.
choisir un ordre $s_1, \ldots, s_i, \ldots, s_{|S|}$ pour le traitement des éléments de S et poser $S' \leftarrow S$
pour $i = 1$ *jusqu'à* $|S|$ **faire**
  soit e l'output de $MIN(S' - \{s_i\}, \emptyset)$;
  si $f_{S'-\{s_i\}}(e) > 0$ **alors** poser $S' \leftarrow S' - \{s_i\}$;

Figure 4.1 – Algorithme de suppression : `Removal`

Nous avons alors la propriété suivante [48].

Proposition 4.1.3. *L'algorithme* `Removal` *produit un IIS en un nombre fini d'étapes.*

Comme l'ensemble S_2 utilisé par la procédure $MIN(S_1, S_2)$ de l'algorithme `Removal` (Figure 4.1) est égal à l'ensemble vide, nous pouvons utiliser un algorithme résolvant SAT pour MIN (à la place d'un algorithme résolvant MaxWSAT). Donc, dans ce

cas-ci uniquement, $MIN(S' - \{s_i\}, \emptyset)$ est un algorithme SAT qui résout le problème du même nom pour la formule CNF induite par $S' - \{s_i\}$ (la formule CNF induite par un ensemble de clauses C' est la formule comprenant les clauses de C' et les variables sur lesquelles agissent les clauses de C', alors que la formule CNF induite par un ensemble de variables $\mathcal{X}'$ est la formule comprenant ces variables et les clauses qui contiennent uniquement des variables de $\mathcal{X}'$). L'output e de $MIN(S' - \{s_i\}, \emptyset)$ est alors VRAI ou FAUX et $f_{S'-\{s_i\}}(VRAI) = 0$ et $f_{S'-\{s_i\}}(FAUX) > 0$.

Afin d'illustrer l'algorithme `Removal`, considérons comme but de trouver un IIS-C dans la formule CNF de la Figure 4.2.

Clause							$\mathcal{M}_1$	$\mathcal{M}_2$	$\mathcal{M}_3$
C_1	$\neg x_1$	$\vee$	x_2				•		•
C_2	$\neg x_1$	$\vee$	$\neg x_2$				•		•
C_3	x_1	$\vee$	x_3				•	•	•
C_4	x_1	$\vee$	$\neg x_3$				•		
C_5	x_3	$\vee$	$\neg x_5$					•	
C_6	$\neg x_3$	$\vee$	$\neg x_5$					•	•
C_7	$\neg x_3$	$\vee$	x_5					•	•
C_8	$\neg x_1$	$\vee$	x_3	$\vee$	$\neg x_4$			•	
C_9	$\neg x_1$	$\vee$	x_4	$\vee$	x_5			•	

Figure 4.2 – Exemple d'une instance SAT non réalisable.

Supposons que les clauses sont enlevées selon le numéro de leur indice.

1. Au départ, $S \leftarrow C$ et $S' \leftarrow S$.
2. La sous-formule induite par $S' - \{C_1\}$ est toujours non réalisable, donc C_1 est enlevé de S', "détruisant" ainsi $\mathcal{M}_1$ et $\mathcal{M}_3$.
3. La sous-formule induite par $S' - \{C_2\}$ est toujours non réalisable, donc C_2 est supprimé de S'.

4. La sous-formule induite par $S' - \{C_3\}$ est réalisable, donc il faut garder C_3.
5. La sous-formule induite par $S' - \{C_4\}$ est non réalisable donc C_4 est supprimé de S'.
6. Ensuite, les sous-formules induites respectivement par $S' - \{C_i\}$ pour $i = 5, 6, 7, 8, 9$ sont réalisables, donc il faut garder ces clauses C_i, et l'IIS-C trouvé est $\mathcal{M}_2$.

Notons que l'ordre dans lequel les clauses ou les variables sont supprimées affecte les résultats de l'algorithme. En effet, si nous avions enlevé les clauses en suivant l'ordre inverse de leur indice, $\mathcal{M}_2$ et $\mathcal{M}_3$ auraient été détruits en premier, et $\mathcal{M}_1$ aurait été retourné.

Illustrons maintenant comment fonctionne l'algorithme `Removal` pour chercher un IIS-V sur ce même exemple de la Figure 4.2. Considérons d'abord l'ordre de traitement des variables selon le numéro de leur indice.

1. Au départ, $S \leftarrow \mathcal{X}$ et $S' \leftarrow S$.
2. La sous-formule induite par $S' - \{x_1\}$ est réalisable, donc la variable x_1 est gardée.
3. La sous-formule induite par $S' - \{x_2\}$ est non réalisable, donc x_2 est supprimé de S'.
4. La sous-formule induite par respectivement $S' - \{x_i\}$ pour $i = 3, 4, 5$ est réalisable, donc ces variables sont conservées. L'IIS-V retourné est $\{x_1, x_3, x_4, x_5\}$.

À nouveau, l'ordre dans lequel les variables sont traitées affecte l'IIS-V obtenu. En effet, si les variables avaient été traitées selon l'ordre inverse du numéro de leur indice, nous aurions obtenu l'IIS-V $\{x_1, x_2, x_3\}$.

Dans la figure 4.3 est présentée la version heuristique de l'algorithme de suppression pour rechercher des IIS. Dans ce cas-ci, `HMIN` est la version heuristique de `MIN`. `HMIN` peut être implémenté en utilisant un algorithme de recherche locale.

HRemoval

Entrée : un ensemble non réalisable S.
Sortie : un sous-ensemble S' qui est possiblement non réalisable.
choisir un ordre $s_1, \ldots, s_i, \ldots, s_{|S|}$ pour le traitement des éléments de S et poser $S' \leftarrow S$
pour $i = 1$ *jusqu'à* $|S|$ **faire**
  soit e l'output de $HMIN(S' - \{s_i\}, \emptyset)$;
  si $f_{S'-\{s_i\}}(e) > 0$ **alors** poser $S' \leftarrow S' - \{s_i\}$;

Figure 4.3 – Heuristique de suppression, HRemoval

Nous avons la propriété suivante [48].

Proposition 4.1.4. *Si l'output de l'algorithme HRemoval est un ensemble non réalisable, alors c'est un IIS.*

Il est possible que le sous-ensemble S' obtenu comme sortie de l'algorithme HRemoval ne soit pas un IIS. Ceci va dépendre de l'heuristique utilisée pour $HMIN$. En effet, nous pouvons illustrer ceci en reprenant l'exemple de la Figure 4.2 que nous avons vu ci-dessus pour trouver un IIS-C. Dans le cas particulier du problème SAT, $HMIN$ est donc une procédure heuristique pour le problème SAT. Imaginons que jusqu'au traitement de la contrainte C_8 la procédure heuristique ne fasse pas d'erreur et que pour la sous-formule induite par $S' - \{C_9\}$ la procédure heuristique retourne FAUX, alors C_9 est supprimé de S' et la sous-formule retournée est réalisable.

4.1.3 L'algorithme Insertion

Le nombre d'étapes (i.e., d'appels à la procédure MIN) requis par l'algorithme Removal afin de trouver un IIS-C ou un IIS-V est égal au nombre de contraintes ou de variables du problème original, sans égard à la taille de son IIS. Ceci n'est pas très efficace dans les cas où l'IIS est beaucoup plus petit que le problème original. Dans de tels cas, il est plus sensé d'utiliser une approche de "bas en haut", où un problème

initialement vide est augmenté jusqu'à ce qu'il devienne non réalisable. L'algorithme `Insertion`, montré dans la Figure 4.4, est une telle approche.

`Insertion`

Entrée : un ensemble non réalisable S.
Sortie : un IIS S'.
poser $S_0 \leftarrow \emptyset$, $T_0 \leftarrow S$ et $i \leftarrow 0$;
répéter
  poser $e_i \leftarrow MIN(S_i, T_i)$;
  si $f_{S_i}(e_i) > 0$ **alors**
    STOP : S_i est un IIS ;
  sinon
    soit s_i un élément de $F_{T_i}(e_i)$;
    poser $S_{i+1} \leftarrow S_i \cup \{s_i\}$, $T_{i+1} \leftarrow T_i \setminus F_{T_i}(e_i)$;
    $i \leftarrow i + 1$;
jusqu'à *ce que l'algorithme s'arrête* ;

Figure 4.4 – Algorithme exact d'insertion, `Insertion`

Proposition 4.1.5. *L'algorithme `Insertion` produit un IIS [48].*

L'algorithme `Insertion` sélectionne un IIS avec un nombre d'étapes égal à la taille de cet IIS.

L'algorithme commence par poser les poids de toutes les clauses ou toutes les variables de S à 1. Ensuite, il modifie ces poids de la façon suivante. Quand nous voulons donner plus d'importance à une clause ou une variable $s \in S$, nous fixons son poids à $\alpha > |S|$, et nous disons que nous rendons s *dure*. Ce sont les variables ou les clauses introduites dans S_{i+1} qui sont durcies et donc ce sont elles qui obtiennent un poids α. D'un autre côté, quand nous ne voulons pas prendre une clause ou un variable s en compte, nous fixons son poids à 0, et nous disons que s est supprimée. Ce sont les variables ou les clauses de $F_{T_i}(e_i) \setminus \{s_i\}$ qui obtiennent un poids nul (car ce sont les variables ou les clauses, sauf s_i, qui sont supprimées de T_i). Ceci incite la procédure `MaxWSAT` à satisfaire toutes les clauses dures de S_i dans le cas de recherche d'un IIS-C ou affecter une valeur à toutes les

variables dures de S_i dans le cas d'un IIS-V, et ignorer toutes celles qui sont supprimées. À chaque itération, `MaxWSAT` retourne une affectation e_i telle que l'ensemble $F_{T_i}(e_i)$ contient au moins une clause ou une variable de chaque IIS restant de $\mathcal{F}$. Ensuite, il durcit une clause ou une variable de $F_{T_i}(e_i)$ et supprime les autres. Quand l'ensemble des clauses ou des variables dures devient non réalisable, `MaxWSAT` obtiendra $f_{S_i}(e_i) > 0$ et l'algorithme retourne cet ensemble S_i.

Illustrons maintenant l'exécution de l'algorithme `Insertion` pour la recherche d'un IIS-C dans la formule CNF de la Figure 4.2.

1. Au départ, $S \leftarrow C$, $S_0 \leftarrow \emptyset$, $T_0 \leftarrow S$. Les poids de toutes les contraintes sont 1.
2. Comme l'ensemble $F_{T_0}(e_0)$ retourné par MaxWSAT doit contenir une contrainte de chacun des trois IIS-C et doit être minimum, il va contenir C_3. Alors, $S_1 = \{C_3\}$ et $T_1 = \{C_1, C_2, C_4, C_5, C_6, C_7, C_8, C_9\}$.
3. L'affectation e_1 doit satisfaire la contrainte C_3 appartenant à S_1 car son poids est α et satisfaire le plus de contraintes de T_1. Supposons que cette affectation e_1 soit $(0, 1, 1, 1, 0)$, alors $F_{T_1}(e_1) = \{C_4, C_7\}$. Choisissons $s_1 = C_7$, donc $S_2 = \{C_3, C_7\}$ et $T_2 = \{C_1, C_2, C_5, C_6, C_8, C_9\}$.
4. L'affectation e_2 doit satisfaire les contraintes de S_2 et satisfaire le plus de contraintes de T_2. Une telle affectation est $e_2 = (0, 1, 1, 1, 1)$. Ainsi, $F_{T_2}(e_2) = \{C_6\}$ et $s_2 = C_6$, donc $S_3 = \{C_3, C_6, C_7\}$ et $T_3 = \{C_1, C_2, C_5, C_8, C_9\}$.
5. L'affectation e_3 doit satisfaire les contraintes de S_3 et satisfaire le plus de contraintes de T_3. Par exemple, e_3 peut être $e_3 = (1, 0, 0, 1, 0)$. Ce qui donne $F_{T_3}(e_3) = \{C_1, C_8\}$. Choisissons $s_3 = C_8$, et ainsi $S_4 = \{C_3, C_6, C_7, C_8\}$ et $T_4 = \{C_2, C_5, C_9\}$.
6. L'affectation e_4 doit satisfaire les contraintes de S_4 et satisfaire le plus de contraintes de T_4, donc par exemple $e_4 = (1, 0, 0, 0, 0)$. Ce qui donne $F_{T_4}(e_4) = \{C_9\}$ et $s_4 = C_9$, $S_5 = \{C_3, C_6, C_7, C_8, C_9\}$ et $T_5 = \{C_2, C_5\}$.

7. L'affectation e_6 doit satisfaire les contraintes de S_5 et satisfaire le plus de contraintes de T_5, donc par exemple $e_5 = (1, 0, 0, 0, 1)$ et $F_{T_5}(e_5) = \{C_5\}$, donc $s_5 = C_5$, $S_6 = \{C_3, C_5, C_6, C_7, C_8, C_9\}$ et $T_6 = \{C_2\}$.
8. À cette étape-là, n'importe quelle affectation e_6 ne satisfait pas au moins une contrainte de S_6 et donc $f_{S_6}(e_6) > 0$ et l'algorithme s'arrête et renvoie S_6. En fait, c'est l'IIS-C $\mathcal{M}_2$ de la Figure 4.2.

Illustrons l'algorithme `Insertion` avec la détection d'un IIS-V dans la formule CNF de la Figure 4.2.

1. Au départ, $S \leftarrow \mathcal{X}$, $S_0 \leftarrow \emptyset$, $T_0 \leftarrow S$. Les poids de toutes les variables sont 1.
2. Comme l'ensemble $F_{T_0}(e_0)$ retourné par `MaxWSAT` doit contenir une variable de chacun des deux IIS-V $\{x_1, x_2, x_3\}$ et $\{x_1, x_3, x_4, x_5\}$, et comme $f_{T_0}(e_0)$ doit être minimum, $F_{T_0}(e_0)$ peut soit contenir x_1 ou x_3, supposons que c'est x_3. Cette variable est ensuite durcie, donc $S_1 = \{x_3\}$ et $T_1 = \{x_1, x_2, x_4, x_5\}$.
3. Le prochain ensemble $F_{T_1}(e_1)$ contient x_1, qui est à son tour durcie. Ainsi, $S_2 = \{x_1, x_3\}$ et $T_2 = \{x_2, x_4, x_5\}$.
4. Le prochain $F_{T_2}(e_2)$ contiendra alors x_2 et une variable du second IIS-V, disons x_4. Nous devons alors choisir de durcir une de ces variables (i.e., l'ajouter à S_2) et supprimer l'autre. Supposons que x_4 est durcie. Alors, $S_3 = \{x_1, x_3, x_4\}$ et $T_3 = \{x_5\}$.
5. Le prochain $F_{T_3}(e_3)$ contient alors x_5. Ainsi, $S_4 = \{x_1, x_3, x_4, x_5\}$ et $T_3 = \{\emptyset\}$. Les variables de S_4 (=variables dures) sont telles que $f_{S_4}(e_4) > 0$, elles forment donc un IIS-V qui est retourné.

À nouveau, le choix de la variable à durcir détermine quel IIS-V sera retourné. Ainsi, si nous avions choisi de durcir x_2 à la place de x_4, l'IIS-V sélectionné précédemment aurait été détruit et un autre aurait été trouvé : $\{x_1, x_2, x_3\}$. Notons finalement, que l'algorithme `Insertion` peut ne pas être capable de trouver tous les IIS d'une formule donnée. En effet, au début de l'algorithme tous les poids valent 1. Si nous considérons le

cas où nous cherchons un IIS de contraintes et que chaque clause de l'IIS de plus petite taille appartient à au moins un autre IIS-C. À la première itération, l'ensemble $F_{T_0}(e_0)$ retourné par `MaxWSAT` contient toutes les clauses de cet IIS de plus petite taille. Une des clauses de cet IIS-C est alors durcie (poids de α) tandis que les autres obtiennent un poids nul et donc cet IIS est détruit. Donc, dans ce cas particulier, cet IIS-C sera toujours détruit lors de la première itération de l'algorithme `Insertion`. Considérons, par exemple, la recherche d'un IIS-C dans la formule CNF de la Figure 4.5. Supposons que toutes les clauses ont le poids 1, l'ensemble $F_{T_0}(e_0)$ optimal, retourné par `MaxWSAT`, clairement contient C_1 et C_2. Comme il contient ces deux clauses, $\mathcal{M}_1$, qui est les plus petit IIS-C de cette formule CNF, sera détruit.

Clause	$\mathcal{M}_1$	$\mathcal{M}_2$	$\mathcal{M}_3$	$\mathcal{M}_4$	$\mathcal{M}_5$
C_1	•	•	•		
C_2	•			•	•
C_3		•		•	
C_4		•		•	
C_5			•		•
C_6			•		•

Figure 4.5 – Un exemple où l'algorithme `Insertion` ne peut pas trouver tous les IIS-C.

Dans la Figure 4.6 est présentée la version heuristique de l'algorithme d'insertion.

Dans ce cas-ci $HMIN$ est une heuristique. Dans le cas du problème SAT c'est une heuristique appelée `MaxWSAT` résolvant le problème Max-SAT pondéré. Comme `MaxWSAT` est une heuristique, il peut arriver que $\mathcal{F}$ devienne réalisable (i.e., $f_{S_i}(e_i) = 0$ et $f_{T_i}(e_i) = 0$), auquel cas un échec est rapporté. L'algorithme peut aussi essayer de réparer cette erreur en utilisant la procédure `Réparer` qui est présentée dans la Figure 4.7. Comme nous l'avons dit, les clauses ou les variables appartenant à S_i ont un poids de α, les clauses ou variables appartenant à T_i ont un poids de 1 et les clauses ou va-

`HInsertion`

Entrée : un ensemble non réalisable S.
Sortie : un sous-ensemble possiblement non réalisable S' ou ERREUR.
poser $S_0 \leftarrow \emptyset$, $T_0 \leftarrow S$ et $i \leftarrow 0$;
répéter
  poser $e_i \leftarrow HMIN(S_i, T_i)$;
  si $f_{S_i}(e_i) > 0$ **alors**
    STOP : S_i est un sous-ensemble qui est possiblement non réalisable ; **retourner** $S' \leftarrow S_i$;
  sinon si $f_{T_i}(e_i) = 0$ **alors**
    STOP : $S_i \cup T_i$ est un ensemble réalisable et nous avons une preuve que l'algorithme s'est trompé à une étape précédente;
    retourner *ERREUR* ou $OK\_REP \leftarrow$ `Réparer` $(T_i, e_i, F_{T_{i-1}}(e_{i-1}) \setminus \{s_{i-1}\})$;
    si OK_REP *est faux* **alors**
      retourner *ERREUR*
  sinon
    soit s_i un élément de $F_{T_i}(e_i)$;
    poser $S_{i+1} \leftarrow S_i \cup \{s_i\}$, $T_{i+1} \leftarrow T_i \setminus F_{T_i}(e_i)$;
    $i \leftarrow i + 1$;
jusqu'à *ce que l'algorithme s'arrête et retourne soit S' soit ERREUR* ;

Figure 4.6 – Heuristique d'insertion, `HInsertion`

riables de $S \setminus \{S_i \cup T_i\}$ ont un poids de 0. La procédure `Réparer` réinsère des clauses ou des variables de $F_{T_{i-1}}(e_{i-1}) \setminus \{s_{i-1}\}$ à T_i (i.e., en remettant leur poids égal à 1), jusqu'à ce que $\mathcal{F}$ devienne à nouveau non réalisable ou jusqu'à ce que toutes les clauses ou variables de $F_{T_{i-1}}(e_{i-1}) \setminus \{s_{i-1}\}$ aient été réinsérées. Les clauses ou les variables que nous réintroduisons sont celles qui, à part s_{i-1}, avaient été enlevées de T_{i-1} (nous ne considérons pas s_{i-1} car son poids a été fixé à α à l'étape précédente). Si `Réparer` ne réussit pas à rendre la formule à nouveau non réalisable, alors la procédure `Réparer` retourne $FAUX$ et l'algorithme `HInsertion` s'arrête.

Proposition 4.1.6. *Si le sous-ensemble obtenu comme sortie de l'algorithme* `HInsertion` *est non réalisable, alors c'est un IIS [48].*

Réparer(T_i, e_i, L)

Entrée : L'ensemble actuel T_i ainsi que l'affectation actuelle e_i et un ensemble L des contraintes ou variables enlevées de T_{i-1} à l'itération précédente.

Sortie : VRAI ainsi que T_i modifié si l'algorithme a réussi à réparer l'erreur, FAUX si l'algorithme n'a pas réussi à réparer son erreur.

$OK \leftarrow VRAI$;
$STOP \leftarrow FAUX$;
tant que $L \neq \emptyset$ *et que* $STOP$ *est* $FAUX$ **faire**
  Choisir $l \in L$;
  $L \leftarrow L \setminus \{l\}$;
  $T_i \leftarrow T_i \cup \{l\}$;
  si $f_{T_i}(e_i) > 0$ **alors**
    $STOP \leftarrow VRAI$;
    retourner $VRAI$ *et* T_i;
si $STOP$ *est* $FAUX$ **alors**
  retourner $FAUX$;

Figure 4.7 – Procédure Réparation qui essaie de réparer une sous-formule réalisable.

4.1.4 L'algorithme HittingSet

L'algorithme HittingSet diffère des deux autres algorithmes parce qu'il trouve des sous-ensembles incohérents minimaux de cardinalité minimum. L'algorithme HittingSet exact est présenté dans la Figure 4.8. Étant donné p ensembles $W_1, \ldots W_p$, nous notons $HS(W_1, \ldots, W_p)$ la procédure qui détermine le plus petit sous-ensemble possible de $W_1 \cup \ldots \cup W_p$ qui intersecte chaque W_i. Cette procédure HS résout le problème du hitting set qui, comme nous l'avons vu à la Section 3.1.1, est NP-difficile.

Proposition 4.1.7. *L'algorithme* HittingSet *produit un IIS de cardinalité minimum [48].*

Bien que l'algorithme HittingSet soit un algorithme fini, son nombre d'itérations peut être exponentiel dans la taille de $|S|$. Mais il peut être stoppé à tout moment afin d'obtenir une borne inférieure sur la taille d'un IIS.

```
HittingSet
Entrée : un ensemble non réalisable S.
Sortie : un IIS.
poser S_0 ← ∅ et i ← 0;
répéter
    poser e_i ← MIN(S_i, S − S_i);
    si f_{S_i}(e_i) > 0 alors
        STOP : S_i est un IIS minimum;
        retourner S_i;
    sinon
        poser i ← i + 1;
        poser S_i ← HS(F_S(e_0), ... F_S(e_{i−1}))
jusqu'à ce que l'algorithme s'arrête ;
```

Figure 4.8 – Algorithme exact `HittingSet`

L'algorithme `HittingSet` est basé sur le fait que, étant donné une formule non réalisable, une solution optimale du problème Max-SAT pondéré, à l'itération i, donne un ensemble $F_S(e_i)$ de clauses non satisfaites ou de variables non affectées qui intersecte chaque IIS. Un IIS $\mathcal{M}$ est par conséquent un *ensemble intersectant* ou *hitting set* en anglais (i.e., un ensemble intersectant chaque ensemble d'une collection) de $\{F_S(e_0), \ldots, F_S(e_i)\}$. Évidemment, un IIS de cardinalité minimum est un *ensemble intersectant minimum* (MHS) de la collection $\{F_S(e_0), \ldots, F_S(e_i)\}$. À chaque itération, la procédure `HS` retourne un MHS S_{i+1} d'une collection $F_S(e_0) \ldots F_S(e_i)$. Les clauses ou variables de S sont alors modifiées de telle sorte que seulement celles dans S_{i+1} deviennent dures. Si l'ensemble $F_{S_i}(e_i)$ retourné par la procédure `MaxWSAT` est non vide, alors S_{i+1} est un IIS de cardinalité minimum et est retourné. Sinon, $F_S(e_i)$ est ajouté à la collection et le même processus est répété.

La Figure 4.9 illustre un exemple d'exécution de l'algorithme `HittingSet` pour la recherche d'un IIS-C dans la formule CNF de la Figure 4.2. Chaque ligne donne le hitting set minimum produit par `HS`, de même que l'affectation optimale e_i trouvée par `MaxWSAT` et l'ensemble $F_S(e_i)$ correspondant, à une itération donnée.

It#	e_i	$F_S(e_i)$	S_{i+1}
0	⟨0 0 0 0 0⟩	$\{C_3\}$	$\{C_3\}$
1	⟨1 0 0 0 0⟩	$\{C_1, C_9\}$	$\{C_3, C_9\}$
2	⟨0 0 1 0 0⟩	$\{C_4, C_7\}$	$\{C_3, C_4, C_9\}$
3	⟨1 0 0 1 0⟩	$\{C_1, C_8\}$	$\{C_1, C_3, C_4\}$
4	⟨1 1 0 1 0⟩	$\{C_2, C_8\}$	$\{C_1, C_2, C_3, C_4\}$
5	⟨0 0 0 0 0⟩	$\{C_3\}$	$f_{S_5}(e_5) > 0$ STOP

Figure 4.9 – Illustration de l'algorithme `HittingSet` sur l'exemple de la Figure 4.2 pour trouver un IIS-C.

1. L'affectation e_0 retournée par `MaxWSAT` viole la contrainte C_3, donc le hitting-set S_1 contient C_3.
2. L'affectation e_1 retournée par `MaxWSAT` viole les contraintes C_1 et C_9, donc le hitting-set minimum S_2 contient C_3 et soit C_1 soit C_9, dans ce cas C_9.
3. L'affectation e_2 viole les contraintes C_4 et C_7, donc le hitting-set minimum S_3 contient C_3 et une contrainte parmi $\{C_1, C_9\}$ ainsi qu'une contrainte parmi $\{C_4, C_7\}$, supposons que $S_3 = \{C_3, C_4, C_9\}$.
4. L'affectation e_3 viole les contraintes C_1 et C_8, et le hitting-set minimum S_4 est formé par C_1, C_3 et une contrainte parmi C_4 ou C_7, supposons C_4.
5. L'affectation e_4 viole les contraintes C_2 et C_8, et le hitting-set minimum S_5 est formé par C_1, C_2, C_3 et C_4.
6. L'affectation e_5 viole C_3 et l'algorithme s'arrête car $f_{S_5}(e_5) > 0$.

La Figure 4.10 illustre l'exécution de l'algorithme `HittingSet` pour la recherche d'un IIS-V dans la formule CNF de la Figure 4.2. Chaque ligne donne le hitting set minimum produit par `HS`, de même que l'affectation optimale e_i trouvée par `MaxWSAT` et l'ensemble $F_S(e_i)$ correspondant, à une itération donnée. Notons que les ensembles $F_S(e_i)$ des trois premières itérations sont identiques à ceux obtenus dans l'exemple donné pour l'algorithme `Insertion`. Le hitting-set minimum de l'itération 3 doit contenir soit x_2 ou x_4, dans cet exemple x_4. Comme avec l'algorithme `Insertion`,

It#	e_i	$F_S(e_i)$	S_{i+1}
0	⟨0 0 - 0 1⟩	$\{x_3\}$	$\{x_3\}$
1	⟨- 1 0 0 0⟩	$\{x_1\}$	$\{x_1, x_3\}$
2	⟨1 - 0 - 0⟩	$\{x_2, x_4\}$	$\{x_1, x_3, x_4\}$
3	⟨1 - 1 0 -⟩	$\{x_2, x_5\}$	$\{x_1, x_2, x_3\}$
4	⟨- 0 0 0 0⟩	$\{x_1\}$	$f_{S_4}(e_4) > 0$ STOP

Figure 4.10 – Illustration de l'algorithme `HittingSet` sur l'exemple de la Figure 4.2 pour trouver un IIS-V.

l'IIS formé des variables x_1, x_2 et x_3 est ensuite détruit. Cependant, l'ensemble $F_S(e_3)$ doit une fois de plus contenir x_2, de telle sorte que le seul hitting set minimum possible à l'itération 4 est l'IIS de cardinalité minimum $\{x_1, x_2, x_3\}$.

Bien que l'algorithme `HittingSet` produise un IIS de cardinalité minimum, (ce qui n'était pas nécessairement le cas pour les algorithmes `Removal` et `Insertion`), il peut avoir besoin d'un nombre exponentiel d'étapes. Par conséquent, cet algorithme convient mieux pour des petites instances.

Bien que l'algorithme `HittingSet` partage certaines similarités avec l'algorithme `CAMUS` de Liffiton et Sakallah [95], des différences fondamentales séparent ces deux méthodes. Premièrement, alors que `CAMUS` se focalise sur l'extraction de tous les IIS, `HittingSet` essaie seulement d'en trouver un avec le plus petit nombre de variables ou clauses. De plus, `CAMUS` requiert de trouver l'ensemble de tous les coMSS, ce qui limite le nombre d'instances qui peuvent être résolues avec cet algorithme. Finalement, `HittingSet` utilise des heuristiques à la place d'un algorithme exact pour le problème SAT, ce qui permet de travailler avec des instances SAT plus difficiles. Des résultats comparatifs peuvent être trouvés dans la Section 4.5 contenant les résultats expérimentaux.

Dans la Figure 4.11 est présentée la version heuristique de l'algorithme de hitting set. Cet algorithme utilise une procédure heuristique $HHS(W_1, \ldots, W_p)$ qui produit un sous-ensemble minimal (au sens de l'inclusion) de $W_1 \cup \ldots \cup W_p$ qui intersecte chaque

W_i.

```
HHittingSet
Entrée : un ensemble non réalisable S.
Sortie : un sous-ensemble S' qui est possiblement un IIS.
poser S_0 ← ∅ et i ← 0;
répéter
    poser e_i ← HMIN(S_i, S − S_i);
    si f_{S_i}(e_i) > 0 alors
        STOP : S_i est possiblement un IIS;
        retourner S_i;
    sinon
        poser i ← i + 1;
        poser S_i ← HHS(F_S(e_0), ... F_S(e_{i−1}))
jusqu'à ce que l'algorithme s'arrête ;
```

Figure 4.11 – Heuristique `HHittingSet`

Proposition 4.1.8. *Si le sous-ensemble obtenu comme sortie de l'algorithme* `HHittingSet` *est un sous-ensemble non réalisable, alors c'est un IIS [48].*

4.2 Autres procédures

Nous pouvons essayer d'accélérer les méthodes que nous avons présentées ci-dessus. Nous avons développé et implémenté des procédures qui vont dans ce sens et qui tentent aussi d'obtenir des IIS plus petits. Dans ce qui suit, nous présentons les différentes procédures que nous avons développées.

4.2.1 L'algorithme `PreFiltering`

Quand nous travaillons avec des grandes instances, il peut être utile de supprimer rapidement autant de variables ou de clauses que possible, en en laissant moins pour l'algorithme de recherche d'un IIS. L'algorithme `PreFiltering`, dont le pseudo-code

est donné dans la Figure 4.12, est une variation de l'algorithme `Insertion`, où toutes les clauses ou variables de $F_{T_i}(e_i)$ sont durcies à chaque itération, i.e., elles sont toutes insérées dans S_{i+1} et leur poids est fixé à α. Ceci permet d'obtenir rapidement un IS de clauses ou de variables sur lequel est ensuite appliqué un algorithme de recherche d'IIS. Comme au moins une clause ou une variable de chaque IIS devient dure à chaque itération, probablement les plus petits IIS vont rester dans la sous-formule CNF $\mathcal{F}$ non réalisable obtenue. L'algorithme `PreFiltering` agit donc comme une heuristique qui tente d'isoler des IIS plus petits. En effet, cet algorithme est un prétraitement supprimant des contraintes ou des variables. Ensuite, une des méthodes de recherche d'IIS est utilisée.

Considérons à nouveau la recherche d'un IIS-C dans la formule CNF de la Figure 4.5, pour laquelle l'algorithme `Insertion` est incapable de trouver l'IIS-C de cardinalité minimum $\mathcal{M}_1$. Le premier ensemble $F_{T_0}(e_0)$ retourné par `MaxWSAT` contient C_1 et C_2. Ces deux clauses sont durcies, de telle sorte que l'algorithme retourne $\mathcal{M}_1$. Par contraste avec l'algorithme `Insertion`, l'algorithme `Prefiltering` combiné avec l'algorithme `Insertion` réussit à trouver l'IIS de cardinalité minimum.

4.2.2 Heuristique basée sur le poids du voisinage

Tout d'abord, nous précisons la notion de densité d'une formule SAT. La densité est définie par le ratio $\frac{m}{n}$ où m est le nombre de clauses et n le nombre de variables. Plus le ratio est grand, plus la densité est définie comme grande.

Rappelons-nous que, dans l'algorithme `Insertion`, le choix de la clause ou de la variable à durcir à chaque itération détermine quel IIS va être obtenu. L'heuristique basée sur le poids du voisinage utilise l'information dans les poids des clauses ou des variables afin de faire des choix qui devraient conduire vers des IIS plus denses et possiblement plus petits. Nous définissons les voisins $\mathcal{N}(C)$ d'une clause $C \in \mathcal{C}$ comme l'ensemble des clauses, excepté C, qui contiennent au moins une des variables contenues dans C. Le

Algorithme 14 : `PreFiltering`

Entrée : Un ensemble non réalisable S ;
Sortie : Un sous-ensemble non réalisable de clauses ou de variables.
Initialisation;
pour $i = 1$ **à** $|S|$ **faire** $w_i \leftarrow 1$;
poser $S_0 \leftarrow \emptyset$, $T_0 \leftarrow S$ et $i \leftarrow 0$;
Construction;
répéter
  poser $e_i \leftarrow MIN(S_i, T_i)$;
  si $f_{S_i}(e_i) > 0$ **alors**
    STOP : S_i est un ensemble non réalisable ;
  sinon
    pour chaque $s_i \in F_{T_i}(e_i)$ **faire**
      poser $S_{i+1} \leftarrow S_i \cup \{s_i\}$;
    $T_{i+1} \leftarrow T_i \setminus F_{T_i}(e_i)$;
    $i \leftarrow i + 1$;
jusqu'à *ce que l'algorithme s'arrête* ;

Figure 4.12 – L'algorithme `PreFiltering`.

poids du voisinage de C est la somme des poids des clauses dans $\mathcal{N}(C)$. De la même façon, les voisins $\mathcal{N}(x)$ d'une variable $x \in \mathcal{X}$ sont composés par l'ensemble des variables, excepté x, qui sont contraintes par au moins une des clauses contenant x. Le poids du voisinage de x est donc la somme des poids des variables dans $\mathcal{N}(x)$.

Quand tous les poids sont égaux (comme par exemple après l'initialisation), le poids du voisinage d'une clause ou d'une variable peut être considéré comme une mesure de la densité de la région autour de cette clause ou variable. Quand les clauses ou variables sont durcies, le poids du voisinage évalue alors la "difficulté" du voisinage. Il est intéressant d'essayer d'extraire des IIS denses, car les IIS les plus denses ont habituellement moins de variables et moins de clauses. En choisissant de rendre dure une clause ou une variable ayant un voisinage de plus grand poids, nous augmentons donc la densité des IIS obtenus. Comme l'algorithme `Insertion` fixe à chaque itération le poids d'une contrainte ou d'une variable à $\alpha > |S|$, l'heuristique de poids de voisinage utilisée

avec l'algorithme `Insertion` va choisir en premier (quand un choix doit être fait) la contrainte ou la variable de plus grand poids de voisinage (ainsi, cela devrait conserver les IIS les plus denses).

Reprenons l'exemple de la Figure 4.2 pour l'illustration de l'algorithme `Insertion` lors de la détection d'un IIS-C donné à la page 95. Dans ce cas-ci la Figure 4.13 donne pour chaque contrainte ses voisins.

contrainte	voisins
C_1	C_2, C_3, C_4, C_8, C_9
C_2	C_1, C_3, C_4, C_8, C_9
C_3	$C_1, C_2, C_4, C_5, C_6, C_7, C_8, C_9$
C_4	$C_1, C_2, C_3, C_5, C_6, C_7, C_8, C_9$
C_5	$C_3, C_4, C_6, C_7, C_8, C_9$
C_6	$C_3, C_4, C_5, C_7, C_8, C_9$
C_7	$C_3, C_4, C_5, C_6, C_8, C_9$
C_8	$C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_9$
C_9	$C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8$

Figure 4.13 – Figure indiquant les voisins de chaque contrainte de l'exemple de la Figure 4.2.

1. Au départ, $S \leftarrow \mathcal{C}$, $S_0 \leftarrow \emptyset$, $T_0 \leftarrow S$. Les poids de toutes les contraintes sont à 1 et les poids de voisinage sont donnés dans le tableau 4.1 où $w_{\mathcal{N}_C}$ est le poids du voisinage de C.

Tableau 4.1 – Les poids de voisinage lorsque $i = 0$.

Contrainte	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9
$w_{\mathcal{N}_C}$	5	5	8	8	6	6	6	8	8

Comme l'ensemble $F_{T_0}(e_0)$ retourné par MaxWSAT doit contenir une contrainte de chacun des trois IIS-C et doit être minimum, il va contenir C_3. Alors, $S_1 = \{C_3\}$ et $T_1 = \{C_1, C_2, C_4, C_5, C_6, C_7, C_8, C_9\}$. Le poids de C_3 est fixé à $\alpha > |C| = 9$

2. Les poids de voisinage à cette étape sont donnés dans le tableau 4.2.

Tableau 4.2 – Les poids de voisinage lorsque $i = 1$

Ctr	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9
$w_{\mathcal{N}_C}$	$4+\alpha$	$4+\alpha$	8	$7+\alpha$	$5+\alpha$	$5+\alpha$	$5+\alpha$	$7+\alpha$	$7+\alpha$

L'affectation e_1 doit satisfaire la contrainte C_3 appartenant à S_1 car son poids est α et satisfaire le plus de contraintes de T_1. Supposons que cette affectation e_1 soit $(0, 1, 1, 1, 0)$, alors $F_{T_1}(e_1) = \{C_4, C_7\}$. Comme le poids de voisinage de C_4 est plus grand que celui de C_7, $s_1 = C_4$ et donc $S_2 = \{C_3, C_4\}$ et $T_2 = \{C_1, C_2, C_5, C_6, C_8, C_9\}$.

3. Les poids de voisinage à cette étape sont donnés dans le tableau 4.3.

Tableau 4.3 – Les poids de voisinage lorsque $i = 2$

Ctr	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9
$w_{\mathcal{N}_C}$	$3+2\alpha$	$3+2\alpha$	$7+\alpha$	$7+\alpha$	$4+2\alpha$	$4+2\alpha$	$4+2\alpha$	$6+2\alpha$	$6+2\alpha$

L'affectation e_2 doit satisfaire les contraintes de S_2 et satisfaire le plus de contraintes de T_2, par exemple $e_2 = (1, 1, 1, 1, 0)$. Ainsi, $F_{T_2}(e_2) = \{C_2\}$ et $S_3 = \{C_2, C_3, C_4\}$ et $T_2 = \{C_1, C_5, C_6, C_8, C_9\}$.

4. Les poids de voisinage à cette étape sont donnés dans le tableau 4.4.

Tableau 4.4 – Les poids de voisinage lorsque $i = 3$

Ctr	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9
$w_{\mathcal{N}_C}$	$2+3\alpha$	$3+2\alpha$	$6+2\alpha$	$6+2\alpha$	$4+2\alpha$	$4+2\alpha$	$5+3\alpha$	$5+3\alpha$	$5+3\alpha$

L'affectation e_3 doit satisfaire les contraintes de S_3 et satisfaire le plus de contraintes de T_3, par exemple $e_3 = (1, 0, 1, 1, 0)$. Ce qui donne, $F_{T_3}(e_3) = \{C_1\}$ et $s_3 = C_1$, donc $S_4 = \{C_1, C_2, C_3, C_4\}$ et $T_2 = \{C_5, C_6, C_8, C_9\}$.

5. L'algorithme s'arrête car n'importe quelle affectation e_4 viole une contrainte de S_4 (i.e., $f_{T_4}(e_4) > 0$) et donc S_4 est un IIS-C, c'est l'IIS-C $\mathcal{M}_1$ de la Figure 4.2, donc c'est l'IIS-C de cardinalité minimum.

Cette stratégie peut aussi être utilisée dans l'algorithme `Removal` en considérant les poids des clauses ou variables supprimées comme étant 0 et les autres étant 1. Comme l'IIS sélectionné a une première clause ou variable supprimée en dernier, nous supprimons ceux avec le plus petit poids de voisinage en premier, ce qui préserve les plus denses. Illustrons ceci, avec les contraintes de la formule CNF de la Figure 4.2.

1. Au départ (i.e., pour $i = 0$), toutes les contraintes ont un poids de 1 et les poids de voisinage pour chaque contrainte sont donnés dans la Figure 4.5.

Tableau 4.5 – Les poids de voisinage lorsque $i = 0$

Contrainte	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9
$w_{\mathcal{N}_C}$	5	5	8	8	6	6	6	8	8

L'algorithme supprime une clause de plus petit poids de voisinage, dans ce cas-ci C_1 ou C_2 sont possibles. Supposons que l'algorithme supprime C_1, donc le poids de C_1 est fixé à 0.

2. Pour $i = 1$, les poids de voisinage sont donnés dans la Figure 4.6 L'algorithme

Tableau 4.6 – Les poids de voisinage lorsque $i = 1$

Contrainte	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9
$w_{\mathcal{N}_C}$	-	4	7	7	6	6	6	7	7

supprime donc C_2 (donc son poids vaut 0).

3. Pour $i = 2$, les poids de voisinage sont donnés dans la Figure 4.7.

Tableau 4.7 – Les poids de voisinage lorsque $i = 2$

Contrainte	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9
$w_{\mathcal{N}_C}$	-	-	6	6	6	6	6	6	6

Comme tous les poids sont égaux, la formule essaie de supprimer une clause parmi celles restantes, disons C_3. Or, la sous-formule obtenue devient réalisable, donc

il faut garder C_3. L'algorithme essaie de supprimer une autre contrainte, disons C_4. Les poids de toutes les contraintes restantes diminuent à 5. Ensuite, quand l'algorithme essaie de supprimer les contraintes C_5 à C_9, la sous-formule devient réalisable, donc ces contraintes doivent être conservées. L'IIS-C ainsi obtenu est $\mathcal{M}_2$ de la Figure 4.2.

Précisons qu'avec cette technique, nous n'obtenons pas nécessairement l'IIS de cardinalité minimum. Nous avons vu clairement, dans la version contraintes de l'exemple ci-dessus (avec `Removal`), que l'IIS-C obtenu n'est pas celui de cardinalité minimum.

4.2.3 Accélération des heuristiques de sélection

Cette section présente une technique qui peut être utilisée pour accélérer la recherche d'IIS quand les versions heuristiques des algorithmes `Insertion` et `PreFiltering` sont utilisées. Dans ces algorithmes, la procédure `MaxWSAT` est une heuristique, plutôt qu'une méthode exacte. Étant donné une formule $\mathcal{F}$, si cette procédure retourne une affectation e_i et un ensemble $F_{T_i}(e_i)$ de contraintes non satisfaites ou de variables non affectées avec $f_{T_i}(e_i) = 1$ (où f retourne le nombre de clauses non satisfaites ou de variables non affectées), nous savons que soit $\mathcal{F}$ est réalisable, soit l'unique contrainte ou variable s de $F_{T_i}(e_i)$ fait partie de l'intersection des IIS. Supposons que $\mathcal{F}$ n'est pas réalisable, la seule clause ou variable dans $F_{T_i}(e_i)$ appartient nécessairement à tous les IIS de $\mathcal{F}$, nous pouvons directement la durcir. De plus, si l'algorithme `MaxWSAT` est implémenté comme une heuristique de recherche locale, un grand nombre d'affectations pour lesquelles $f_{T_i}(e_i) = 1$ peuvent être visitées. Nous pouvons ainsi mémoriser les clauses insatisfaites ou les variables non affectées correspondant à ces affectations, et les durcir toutes une fois que la recherche locale est terminée. Cette technique est particulièrement efficace quand $\mathcal{F}$ contient un seul IIS. Dans de tels cas, une seule itération de l'algorithme de recherche d'IIS (donc un seul appel à `MaxWSAT`) suffit souvent pour trouver l'IIS en entier. Cette idée peut aussi être utilisée pour prouver que les formules non

réalisables obtenues en utilisant l'algorithme `Insertion` ou `PreFiltering` sont minimales. Par conséquent, si `MaxWSAT` trouve $f_{T_i}(e_i) = 1$ à chaque itération de l'algorithme de recherche d'IIS (en excluant la dernière itération où $f_{T_i}(e_i) \geq \alpha$), nous savons que soit la sous-formule trouvée est le seul IIS du problème, soit elle est réalisable.

4.3 Algorithme tabou pour MaxWSAT

Dans les algorithmes de sélection, les procédures `SAT` et `MaxWSAT` peuvent être exactes ou heuristiques. Cependant, comme le problème SAT est NP-complet et le problème Max-SAT pondéré est NP-difficile, les algorithmes exacts peuvent avoir certaines difficultés à résoudre des grandes instances, et, dans ces cas-là, nous devons nous tourner vers des heuristiques. Afin de résoudre ces problèmes de façon efficace, nous avons implémenté une heuristique de recherche locale basée sur l'algorithme de recherche tabou [62]. Desrosiers et al. [39] ont utilisé avec succès de telles heuristiques pour trouver des sous-graphes sommets-critiques et arêtes-critiques dans le cas du problème de k-coloration pour des grandes instances. Lors de nos expériences avec l'algorithme `Removal`, nous avons utilisé une procédure `SAT` exacte : l'algorithme `zChaff` de Moskewicz et al. [107]. Par contre, pour les algorithmes `Insertion` et `HittingSet`, nous avons utilisé des procédures `MaxWSAT` heuristiques utilisant la recherche tabou. En effet, pour le problème SAT il existe de nombreux algorithmes exacts très performants ; par contre, pour le problème Max-SAT pondéré, il existe moins d'algorithmes exacts et ils ne sont pas aussi performants que ceux existants pour SAT.

Pour la détection d'IIS de contraintes, l'algorithme tabou pour `MaxWSAT` est présenté dans la Figure 4.14. Nous utilisons comme espace des solutions $\mathcal{S}$ l'ensemble des affectations complètes possibles notées $s : \mathcal{X} \rightarrow \{0, 1\}$ (nous employons cette lettre $\mathcal{S}$ pour désigner l'ensemble des solutions, car c'est la notation qui est principalement utilisée dans la littérature sur les méthodes de recherche tabou ; cependant, il faut faire

attention de ne pas confondre avec le terme S des algorithmes de détections d'IIS, qui peut être un ensemble de contraintes ou de variables). La fonction de coût $f_w(s)$ servant à mesurer la qualité d'une affectation s est la somme des poids des clauses insatisfaites par s, que nous minimisons. Si, à une certaine itération, plusieurs mouvements mènent à des affectations de même coût minimum, un parmi ceux-ci est choisi au hasard. Étant donné une affectation complète s, un voisin de s est obtenu en inversant la valeur de vérité d'une seule variable. Quand nous cherchons le meilleur voisin s' de s, nous ne recalculons pas le coût de chaque voisin. En effet, pour connaître en temps constant le coût de chaque mouvement, nous maintenons à jour une matrice $Gamma$ de taille nombre de variables par deux (valeurs 0 et 1) telle que $Gamma(x, i)$ stocke le coût de l'affectation de i à x. Lorsqu'un mouvement est effectué qui modifie la valeur de x en changeant $s(x)$ par $s'(x)$, l'augmentation ou la diminution du coût actuel est donné par le calcul suivant $Gamma(x, s'(x)) - Gamma(x, s(x))$. Quand un mouvement $(x', s'(x'))$ est effectué, la ligne de la variable x' de la matrice $Gamma$ est mise à jour. De plus, afin d'éviter de cycler, pour s'échapper des minima locaux, l'algorithme de recherche tabou interdit, durant τ itérations, de changer la valeur de la variable affectée, à moins que cela améliore le meilleur coût trouvé à ce moment. Nous avons implémenté trois façons différentes de calculer τ. Premièrement, il est possible de lui attribuer une valeur fixe pour toutes les itérations. Deuxièmement, il est possible de donner deux valeurs τ_{min} et τ_{max} et à chaque itération τ est choisi au hasard dans l'intervalle $[\tau_{min}, \tau_{max}]$. Troisièmement, il est possible de choisir τ au hasard dans l'intervalle $[\lambda\sqrt{NV}, 2\lambda\sqrt{NV} - 1]$ où λ est un paramètre et NV est le nombre de contraintes violées. Pour les résultats expérimentaux que nous présentons dans la Section 4.5, nous avons utilisé la troisième méthode de calcul pour τ avec $\lambda = 1.5$ ou 3.0. Afin de savoir si un mouvement $(x, s(x))$ est tabou, nous conservons dans $\Lambda(x, s(x))$ l'itération à partir de laquelle le mouvement n'est plus tabou. À chaque fois qu'un mouvement $(x, s(x))$ est effectué, alors la valeur de $\Lambda(x, s(x))$ est mise à jour et vaut le numéro de l'itération courante + τ. La valeur retournée par `MaxWSAT` est une affectation complète des variables s^* et l'ensemble U

de clauses non satisfaites par la solution s^* tel que $f_w(s^*)$ est la plus petite valeur de la fonction objectif rencontrée. L'algorithme s'arrête soit après un nombre maximal fixé d'itérations ou un nombre maximal d'itérations sans amélioration ou après un temps d'exécution cpu maximal.
La procédure expliquant comment U est mise à jour est présentée dans la Figure 4.15 : si une contrainte C était non satisfaite au début de l'itération et devient satisfaite par le changement de la valeur de x', alors C est supprimée de U ; dans le cas contraire où C était satisfaite au début de l'itération et devient non satisfaite par le changement de la valeur de x alors C est ajoutée à U.

D'un autre côté, pour la détection des IIS de variables, la procédure tabou pour `MaxWSAT` est présentée dans la Figure 4.16. Dans ce cas-ci, l'espace des solutions $\mathcal{S}$ de `MaxWSAT` est l'ensemble de toutes les affectations partielles légales $s : \mathcal{X} \rightarrow \{0, 1, -\}$, où "-" signifie qu'il est possible qu'une variable ne soit pas affectée. La fonction de coût $f_w(s)$ calcule la somme des poids des variables non affectées et le but est de minimiser $f_w(s)$. Une solution voisine de s est obtenue en donnant une valeur à une variable non affectée de s, et si nécessaire en désaffectant des variables appartenant à des clauses insatisfaites, en commençant avec celles qui sont impliquées dans le plus de clauses insatisfaites, jusqu'à ce que l'affectation soit évaluée à vraie.

Ce processus d'affectation et de désaffectation est effectué par la procédure `Affectation` dont le pseudo-code est donné dans la Figure 4.17. Dans cette procédure, nous notons $\mathcal{X}_C$ l'ensemble des variables de $\mathcal{X}$ participant à la contrainte C. Lorsqu'une affectation d'une variable x est testée ou est effectuée, il faut parcourir les contraintes et mémoriser les contraintes auxquelles participe x qui ne sont pas satisfaites (i.e., où toutes les variables sont affectées et dont aucune variable ne satisfait la contrainte). Le processus de désaffectation s'effectue de manière gloutonne en désaffectant d'abord les variables impliquées dans le plus de contraintes non satisfaites. En cas d'égalité entre plusieurs variables possibles pour la désaffectation, une des variables

MaxWSAT($\mathcal{F}, W$) tabou pour une affectation complète des variables

Entrée : Une formule CNF $\mathcal{F}$ non réalisable et l'ensemble W des poids des clauses.
Sortie : Une affectation complète des variables s^* et un sous-ensemble U de clauses non satisfaites.

Initialisation
pour chaque $x \in \mathcal{X}$ **faire**
  $s(x) \leftarrow$ attribution aléatoire de la valeur 0 ou 1;
  $\Lambda(x,0) \leftarrow 0$ et $\Lambda(x,1) \leftarrow 0$;
$s^* \leftarrow s$ et $iter \leftarrow 1$ et $U \leftarrow \emptyset$;
pour chaque $C \in \mathcal{C}$ **faire**
  si *C n'est pas satisfaite* **alors** $U \leftarrow U \cup \{C\}$;

Tabou
tant que *aucun critère d'arrêt n'est rencontré* **faire**
  $ListeVar \leftarrow \emptyset$ et $f_{best} \leftarrow +\infty$;
  pour chaque *variable x appartenant à une clause de U* **faire**
    $s' \leftarrow s$;
    $s'(x) \leftarrow 1 - s(x)$;
    si $iter > \Lambda(x, s'(x))$ **ou** $f_w(s') < f_w(s^*)$ **alors**
      si $f_w(s') < f_{best}$ **ou** $ListeVar = \emptyset$ **alors**
        $ListeVar \leftarrow \{x\}$;
        $f_{best} \leftarrow f_w(s')$;
      sinon si $f_w(s') = f_{best}$ **alors**
        $ListeVar \leftarrow ListeVar \cup \{x\}$;
  si $ListeVar = \emptyset$ **alors**
    $x' \leftarrow$ choisir au hasard une variable appartenant à une clause de U;
  sinon
    $x' \leftarrow$ choisir au hasard une variable appartenant à $ListeVar$;
  $s(x') \leftarrow 1 - s(x)$;
  $\Lambda(x', s(x')) \leftarrow iter + \tau$;
  si $f_w(s) < f_w(s^*)$ **alors** $s^* \leftarrow s$;
  MettreAJour $(U, \mathcal{C}, (x', s(x'))$;
  $iter \leftarrow iter + 1$;
retourner U;

Figure 4.14 – Algorithme tabou pour MaxWSAT pour une affectation complète des variables (lors de la recherche d'IIS de contraintes).

`MettreAJour`(U,$\mathcal{C}$,$(x', s(x'))$)

Entrée : L'ensemble U des clauses non satisfaites, l'ensemble $\mathcal{C}$ de toutes les clauses et la variable changée x' ainsi que sa valeur $s(x')$.
Sortie : L'ensemble U mis à jour

pour chaque $C \in \mathcal{C}$ *contenant* x' **faire**
  si $C \in U$ *et* C *est maintenant satisfaite par* x' **alors**
    $U \leftarrow U \setminus \{C\}$;
  sinon si $C \notin U$ *et* C *est maintenant non satisfaite par* x' **alors**
    $U \leftarrow U \cup \{C\}$;

Figure 4.15 – Procédure mettant à jour U.

est tirée au hasard. Ce processus est répété jusqu'à ce que toutes les clauses soient satisfaites. Dans notre implémentation pratique de cet algorithme tabou, cette procédure `Affectation` n'est pas appelée pour tester les coûts de tous les mouvements à chaque itération, mais elle est appelée une seule fois à la fin de l'algorithme quand le mouvement est effectué. En effet, afin de connaître le coût de chaque mouvement en temps constant, nous maintenons à jour une matrice appelée $Gamma$ de taille nombre de variables par trois (valeurs 0, 1 ou $-$ quand une variable n'est pas affectée) telle que $Gamma(x, i)$ stocke le coût de l'affectation de i à x. Le calcul $Gamma(x, s'(x)) - Gamma(x, s(x))$ donne le coût du mouvement consistant à affecter $s'(x)$ à x à la place de $s(x)$. Quand un mouvement est effectué, la ligne de la variable affectée ainsi que les lignes de la ou des variables désaffectées de la matrice $Gamma$ sont mises à jour.

Une fois de plus, la recherche tabou interdit, durant τ itérations, de donner à une variable non affectée la valeur qu'elle vient juste de perdre, à moins que cela améliore le meilleur coût trouvé. Comme pour la recherche d'IIS de contraintes, nous avons implémenté les trois mêmes façons de calculer la valeur de τ (dans le troisième cas, NV est le nombre de variables non affectées). À nouveau, $\Lambda(x, s(x))$ stocke l'itération à partir de laquelle le mouvement n'est plus tabou. La valeur retournée par `MaxWSAT` est une affectation partielle des variables s^* et l'ensemble U des variables non affectées dans s^* tel que

MaxWSAT($\mathcal{F}, W$) tabou pour une affectation partielle de variables

Entrée : Une formule CNF $\mathcal{F}$ non réalisable et l'ensemble W des poids des variables.
Sortie : Une affectation partielle des variables s^* et le sous-ensemble U des variables non affectées de s^*.

Initialisation
$U \leftarrow \emptyset$;
pour chaque $x \in \mathcal{X}$ **faire**
  $s(x) \leftarrow$ "-" : aucune variable ne reçoit une valeur ;
  $\Lambda(x,0) \leftarrow 0$ et $\Lambda(x,1) \leftarrow 0$ et $U \leftarrow U \cup \{x\}$;
$s^* \leftarrow s =$ et $iter \leftarrow 1$;

Tabou
tant que *aucun critère d'arrêt n'est rencontré* **faire**
  $ListeMvt \leftarrow \emptyset$ et $f_{best} \leftarrow +\infty$;
  pour chaque *variable* $x \in U$ **faire**
    pour chaque $y \in \{0,1\}$ **faire**
      $s' \leftarrow$ `Affectation` $(\mathcal{F}, w, s, (x,y))$;
      si $iter > \Lambda(x,y)$ **ou** $f_w(s') < f_w(s^*)$ **alors**
        si $f_w(s') < f_{best}$ **ou** $ListeMvt = \emptyset$ **alors**
          $ListeMvt \leftarrow \{(x,y)\}$;
          $f_{best} \leftarrow f_w(s')$;
        sinon si $f_w(s') = f_{best}$ **alors**
          $ListeMvt \leftarrow ListeMvt \cup \{(x,y)\}$;
  si $ListeMvt = \emptyset$ **alors**
    $x' \leftarrow$ choisir au hasard une variable non affectée;
    $y' \leftarrow$ choisir au hasard une valeur parmi 0 ou 1;
  sinon $(x', y') \leftarrow$ choisir au hasard un mouvement appartenant à $ListeMvt$;
  $s' \leftarrow$ `Affectation` $(\mathcal{F}, U, w, s, (x', y'))$;
  $U \leftarrow U \setminus \{x'\}$;
  pour chaque *variable* x *désaffectée par la procédure* `Affectation` **faire**
    $\Lambda(x, s(x)) \leftarrow iter + \tau$;
    $U \leftarrow U \cup \{x\}$;
  $s \leftarrow s'$;
  si $f_w(s) < f_w(s^*)$ **alors** $s^* \leftarrow s$;
  $iter \leftarrow iter + 1$;
retourner U;

Figure 4.16 – Algorithme tabou pour MaxWSAT pour une affectation partielle des variables (lors de la recherche d'IIS de variables).

```
Affectation(F, U, w, s, (x, y))
Initialisation
L ← ∅;
score(v) = 0 ∀ variable v;
s(x) = y;
pour chaque contrainte C tq x ∈ X_C faire
    si C n'est plus satisfaite alors L ← L ∪ {C};
pour chaque variable v ∈ X \ U tq v ≠ x faire
    pour chaque C ∈ L faire
        si v ∈ X_C alors score(v) ← score(v) + 1
tant que L ≠ ∅ faire
    Choisir v tq score(v) est maximum;
    s(v) = −;
    score(v) = 0;
    pour chaque C tq v ∈ X_C faire
        L ← L \ {C};
        pour chaque w ∈ X_C tq w ≠ v faire
            score(w) ← score(w) − 1;
```

Figure 4.17 – Procédure `Affectation`.

$f_w(s^*)$ est la plus petite valeur de la fonction objectif rencontrée.

À nouveau, l'algorithme s'arrête soit après un nombre maximal fixé d'itérations ou un nombre maximal d'itérations sans amélioration ou après un temps d'exécution cpu maximal.

Nous présentons maintenant un exemple de déroulement de l'algorithme tabou pour rechercher un IIS-V sur l'exemple de la Figure 4.2. Entre parenthèses se trouve d'abord l'affectation actuelle, puis nous donnons le coût actuel et le coût des mouvements possibles. Supposons que la longueur tabou est fixe et vaut 4 dans cet exemple. Nous montrons l'initialisation et les sept premières itérations de l'algorithme

- Initialisation : (- - - - -) : Toutes les variables sont non affectées, donc le coût actuel vaut 5. Tous les mouvements possibles ont un coût de -1. Choisissons le

mouvement (5,0), i.e., d'attribuer la valeur 0 à la variable x_5.

- Itération 1 : (- - - - 0) : Le coût total vaut 4. Tous les mouvements possibles ont un coût de -1 sauf (3,1) qui a un coût de 0 (en effet si nous mettions x_3 à vrai, il faudrait désaffecter x_5 car la contrainte C_7 ne serait plus satisfaite). Le mouvement choisi est (1,1).
- Itération 2 : (1 - - - 0) : Le coût total vaut 3. Les mouvements (4,1) et (3,0) ont un coût de -1 et les mouvements (2,0), (2,1), (4,0) et (3,1) ont un coût de 0 car il faudrait désaffecter une variable. Choisissons le mouvement (4,1).
- Itération 3 : (1 - - 1 0) : Le coût total vaut 2. Les mouvements (2,0), (2,1), (3,0) et (3,1) ont un coût de 0 car il faut désaffecter une variable. Le mouvement choisi est (3,0), une seule contrainte devient non satisfaite : la contrainte C_8, alors il faut désaffecter une variable de cette contrainte, soit la variable x_1 soit la variable x_5 et le choix se fait au hasard. Choisissons x_1 et alors le mouvement (1,1) est rendu tabou jusqu'à l'itération 7.
- Itération 4 : (- - 0 1 0) : Le coût total vaut 2. Les mouvements (1,0) et (1,1) (tabou) sont de coût 0 et les mouvements (2,0) et (2,1) sont de coût -1. le mouvement (2,1) est choisi.
- Itération 5 : (- 1 0 1 0) : Le coût total est 1. Le mouvement (1,1) est tabou, le seul mouvement possible est alors (1,0) de coût 0 car il faut désaffecter la variable x_3 car sinon la contrainte C_3 serait non satisfaite. Alors le mouvement (3,0) est rendu tabou jusqu'à l'itération 9.
- Itération 6 : (0 1 - 1 0) : de coût total de 1. Le mouvement (3,0) est tabou. Le seul mouvement possible est (3,1) de coût 1. En effet, les contraintes C_4 et C_7 sont non satisfaites, comme elles n'ont aucune variable en commun il faut désaffecter deux variables : x_1 et x_5. Les mouvements (1,0) et (5,0) sont rendus tabou jusqu'à l'itération 10.
- Itération 7 : (- 1 1 1 -) : de coût total 2. Le seul mouvement non tabou possible est (5,1) de coût 0. La contrainte C_6 devient alors non satisfaite et il faut désaffecter

x_3. Donc le mouvement (3,1) est rendu tabou jusqu'à l'itération 11.

4.4 Détails d'implémentation

Nous utilisons un algorithme exact afin de résoudre le problème de hitting set minimum (procédure `HS` de la Figure 4.8). En effet, ce problème de hitting set minimum est formulé en termes d'un programme linéaire en nombres entiers et est résolu à l'aide de `CPLEX`.

$$
\begin{aligned}
min \quad & \sum_{i=1}^{m} x_i \\
s.t. \quad & \\
& \sum_{i \in U_j} x_i \geq 1, \quad 1 \leq j \leq n && (4.1) \\
& \sum_{i=1}^{m} x_i \geq K && (4.2) \\
& x_i \in \{0,1\}, \quad 1 \leq i \leq m
\end{aligned}
$$

Les U_j sont les sous-ensembles $F_S(e_j)$ produits par l'affectation e_j obtenue par la procédure `MIN` de la Figure 4.8, K est la cardinalité du hitting set obtenu à l'itération précédente de l'algorithme `HittingSet`, et x_i est égal à 1 si et seulement si la i-ème clause ou variable est incluse dans le hitting set. Le rôle de la contrainte (4.2) est d'accélérer la recherche en réduisant la taille de l'espace des solutions, car il ne peut pas y avoir de hitting set avec strictement moins que K éléments.

4.5 Résultats expérimentaux

Dans cette section, nous présentons les expériences faites pour évaluer nos algorithmes de sélection d'IIS en utilisant différentes instances aléatoires ou provenant de problèmes réels. Nous avons testé trois méthodes différentes.

- Dans la première méthode, nommée "`Removal`", nous utilisons d'abord l'algorithme exact `Removal` pour trouver un IIS-V en supprimant des variables dans un ordre aléatoire. Ensuite, nous supprimons des clauses de cet IIS-V, en utilisant la version "contraintes" de l'algorithme `Removal`, pour trouver un IIS-C.
- Dans la deuxième méthode, nommée "`P+Insertion`", nous utilisons d'abord l'algorithme `PreFiltering` afin de réduire le nombre de variables. Nous utilisons ensuite l'algorithme `Insertion` afin de trouver un IIS-V, et répétons le même processus sur l'IIS-V, cette fois pour trouver un IIS-C. Nous utilisons un des deux algorithmes de recherche tabou décrits à la section 4.3 : si nous cherchons un IIS-V c'est l'algorithme présenté dans la Figure 4.16 et si nous cherchons un IIS-C c'est l'algorithme présenté dans la Figure 4.14, pour résoudre le problème Max-SAT pondéré. De plus, nous utilisons l'heuristique basée sur le poids du voisinage pour sélectionner les clauses et variables à insérer dans l'IIS, et aussi la technique d'accélération décrite à la Section 4.2.3 afin d'accélérer l'extraction. Parce que cette méthode utilise une heuristique pour résoudre le problème Max-SAT pondéré, elle est encline à faire des erreurs qui peuvent rendre le sous-ensemble $S_i \cup T_i$ de la Figure 4.6 de clauses ou variables réalisable. Dans ces cas-là, nous utilisons la procédure `Réparer` de la Figure 4.7 qui ré-introduit aléatoirement des clauses ou variables supprimées précédemment jusqu'à ce que le sous-ensemble $S_i \cup T_i$ redevienne non réalisable, i.e., jusqu'à ce que $f_{S_i \cup T_i}(e_i) > 0$, ou jusqu'à ce qu'elle ait la preuve qu'elle ne puisse pas réparer.
- Dans la troisième méthode, nous trouvons des IIS de cardinalité minimum en utilisant l'algorithme `HittingSet` avec un des algorithmes de recherche tabou (se-

lon qu'il s'agit d'un IIS-C ou d'un IIS-V) pour résoudre le problème Max-SAT pondéré et l'algorithme exact décrit à la Section 4.4 pour résoudre le problème de hitting set minimum. Contrairement aux deux autres méthodes, nous trouvons des IIS de cardinalité minimum de variables et clauses séparément, et écrivons "`HS-V`" et "`HS-C`", la méthode `HittingSet` qui trouve respectivement des IIS-V ou des IIS-C.

Nous comparons ces trois méthodes avec plusieurs des algorithmes populaires existants pour extraire des IIS-C. Premièrement, `zCore` de Zhang et Malik [140] et `AMUSE` de Oh et al. [109] sont des algorithmes disponibles sur internet qui produisent des sous-formules non réalisables (non nécessairement minimales). Ils peuvent être combinés avec l'algorithme `zMinimal` de Zhang et Malik [140] (aussi disponible sur internet) qui est similaire à notre algorithme `Removal` pour trouver des IIS-C. Nous utilisons la sortie de `zCore` et `AMUSE` comme entrée de `zMinimal`. La combinaison de `zCore` et `AMUSE` avec `zMinimal` est notée `zCore+` et `AMUSE+`.

Nous comparons aussi nos résultats avec ceux obtenus en utilisant les algorithmes `MUP` de Huang [78], `SMUS` de Mneimneh et al. [106] et `CAMUS` de Liffiton et al. [95] qui produisent tous des IIS-C. L'algorithme `SMUS` offre en plus la garantie que l'IIS-C obtenu en sortie est de cardinalité minimum (en nombre de clauses). De plus, comme l'algorithme `CAMUS` extrait tous les IIS-C d'une formule CNF, il trouve aussi celui avec le nombre minimum de clauses.

Les algorithmes proposés par Bruni [26] et Mazure et al. [101] trouvent des sous-formules non réalisables non nécessairement minimales. Comme ils ne sont pas disponibles sur internet, nous ne pouvons pas combiner la sous-formule obtenue en sortie avec `zMinimal`.

Toutes les expérimentations ont été effectuées sur un PC 2GHz Intel Pentium IV avec 512kB de cache et 1 GB de RAM, sous Linux CentOS release 4.2. Tous les temps d'exécution sont en secondes à part les valeurs de temps finissant par un "**h**", dans ces cas-là les valeurs sont en heures. Pour nos méthodes, nous avons effectué dix relances en uti-

lisant des graines différentes, et nous donnons les valeurs moyennes des relances qui ont réussi. Pour les algorithmes `HS-V` et `HS-C`, toutes les valeurs qui ne sont pas des bornes inférieures (i.e., qui ne sont pas précédées par le signe "$\geq$") correspondent à 10 réussites pour chaque instance. Pour la méthode `Removal`, comme nous employons une version exacte le nombre de réussites est de 10 pour chaque instance. Pour la méthode `P+Insertion`, le nombre de réussites va être précisé pour chaque type d'instances. Toutes les valeurs rapportées pour `zCore+` et `AMUSE+` ont été obtenues sur notre ordinateur (puisque ces algorithmes peuvent être téléchargés sur internet). Pour les autres algorithmes, les solutions et les temps rapportés sont pris des articles correspondants (quand ils sont disponibles). Bruni a effectué ses expérimentations sur un PC Pentium 450MHz (beaucoup plus lent que notre ordinateur), Mazure et al. sur un PC Pentium 133MHz (beaucoup plus lent que notre ordinateur), `MUP` a été testé sur un pentium IV 2.4GHz (un peu plus rapide que notre ordinateur), `SMUS` sur un Pentium IV 2GHZ (équivalent à notre ordinateur) et `CAMUS` sur un Opteron 2.2GHZ avec 8GB de RAM (un petit peu plus rapide que notre ordinateur, mais surtout avec huit fois plus de mémoire vive). Dans les tableaux de résultats qui suivent, "**Nom**" donne le nom de l'instance testée, "**V**" et "**C**" sont le nombre de variables et de clauses de l'instance originale ou de l'IIS. De plus, "**C'**" réfère au nombre de clauses de l'IIS-C obtenu après avoir réduit un IIS-V, et "**t**" donne le temps CPU.

4.5.1 Instances DIMACS

Le premier ensemble d'instances, qui peut être trouvé sur le site ftp du Dimacs [1], provient du deuxième Challenge DIMACS. Ces instances sont de quatre types différents. Les instances *AIM*, fournies par Asahiro et al. [9], sont des instances 3-SAT artificielles. Les instances *JNH*, fournies par Hooker, sont des instances aléatoires difficiles générées en rejetant les clauses unitaires et en fixant la densité (ratio du nombre de clauses par le nombre de variables) à une valeur difficile : comme nous l'avons vu dans le chapitre 2,

le seuil autour duquel les instances sont difficiles est 4.24 pour les instances de 3-SAT. Finalement, les instances *SSA* et *BF*, fournies par Van Gelder et Tsuji, sont des instances provenant de problèmes réels d'analyse d'erreurs dans des circuits.

Instances AIM Les instances AIM sont relativement petites et n'ont qu'un IIS-C à part deux d'entre elles qui ont deux IIS-C de même taille. Pour ces instances, nous avons trouvé des IIS de variables et de clauses avec les algorithmes `Removal`, `P+Insertion`, `HS-V` et `HS-C`. Les résultats sont présentés dans le Tableau 4.8 et sont comparés avec ceux obtenus par `zCore+`, `AMUSE+`, Mazure et al. et par Bruni. Pour chacune des instances AIM, toutes les dix relances de la méthode `P+Insertion` ont réussi. Nous observons que nos méthodes trouvent des IIS-C avec un nombre plus petit ou égal de variables et clauses que les sous-ensembles incohérents (pas forcément minimaux) trouvés par les autres algorithmes. Pour l'instance *aim100-1.6-2*, Mazure et al. trouvent un IIS-C avec une clause de moins que nous. Nous sommes sûrs que ceci est une erreur car `HS-C` a prouvé que notre IIS-C ayant 53 clauses est minimum en termes de cardinalité. Nous pouvons remarquer que dans presque tous les cas, les IIS sont obtenus de façon très rapide. Par exemple, tant la méthode `Removal` que `HS-C` trouvent des IIS-C en moins de 3 secondes. La méthode `P+Insertion` est légèrement plus lente que `Removal`. Ceci est dû au fait que `zChaff` résout ces instances de manière exacte en très peu de millisecondes, beaucoup plus rapidement que le temps requis par notre algorithme de recherche tabou. En plus, nous observons que les algorithmes `zCore+` et `AMUSE+` sont nettement plus rapides que les autres algorithmes.

Instances JNH Contrairement aux instances AIM, les instances JNH ont généralement plusieurs IIS-C. Pour ces instances, nous avons testé les méthodes `Removal`, `P+Insertion`, `HS-V`, `HS-C` avec une limite de temps d'une heure, et nous avons comparé nos résultats avec ceux obtenus par Bruni et avec `zCore+` et `AMUSE+`. Les

Tableau 4.8 – Résultats pour les instances AIM.

Instance			zCore+			AMUSE+			Mazure			Bruni			Removal				P+Insertion				HS-C			HS-V		
Nom	V	C	V	C	t	V	C	t	V	C	t	V	C	t	V	C	C'	t	V	C	C'	t	V	C	t	V	C	t
aim50-1.6-1	50	80	20	22	0.0	20	22	0.1	20	22	0.1	20	22	0.1	20.0	22.0	22.0	0.2	20.0	22.0	22.0	1.7	20.0	22.0	0.1	20.0	22.0	0.1
aim50-1.6-2	50	80	28	32	0.0	28	32	0.0	28	32	0.1	28	32	0.1	28.0	34.0	32.0	0.3	28.0	34.0	32.0	2.7	28.0	32.0	0.2	28.0	34.0	0.2
aim50-1.6-3	50	80	28	31	0.0	28	31	0.1	28	31	0.1	28	31	0.1	28.0	33.0	31.0	0.3	28.0	33.0	31.0	3.5	28.0	31.0	0.2	28.0	33.0	0.3
aim50-1.6-4	50	80	18	20	0.0	18	20	0.0	18	20	0.1	18	20	0.0	18.0	21.0	20.0	0.2	18.0	21.0	20.0	1.9	18.0	20.0	0.1	18.0	21.0	0.1
aim50-2.0-1	50	100	21	22	0.0	21	22	0.0	21	22	0.1	21	22	0.1	21.0	26.0	22.0	0.3	21.0	26.0	22.0	1.8	21.0	22.0	0.1	21.0	26.0	0.2
aim50-2.0-2	50	100	28	30	0.0	28	30	0.1	28	30	0.1	28	31	0.3	28.0	34.0	30.0	0.3	28.0	34.0	30.0	5.4	28.0	30.0	0.2	28.0	34.0	0.6
aim50-2.0-3	50	100	22	28	0.0	22	28	0.1	22	28	0.1	22	28	0.0	22.0	29.0	28.0	0.2	22.0	29.0	28.0	2.4	22.0	28.0	0.2	22.0	29.0	0.3
aim50-2.0-4	50	100	18	21	0.0	18	21	0.1	18	21	0.1	18	21	0.3	18.0	23.0	21.0	0.2	18.0	23.0	21.0	1.7	18.0	21.0	0.1	18.0	23.0	0.2
aim100-1.6-1	100	160	43	47	0.1	43	47	0.1	43	47	0.2	43	47	1.2	43.0	49.0	47.0	0.7	43.0	49.0	47.0	5.1	43.0	47.0	0.3	43.0	49.0	0.7
aim100-1.6-2	100	160	46	53	0.1	46	53	0.1	46	52	0.3	46	54	4.5	46.0	57.0	53.0	0.8	46.0	57.0	53.0	6.0	46.0	53.0	0.4	46.0	57.0	1.5
aim100-1.6-3	100	160	51	57	0.1	51	57	0.1	51	57	0.3	51	57	4.6	51.0	59.0	57.0	0.8	51.0	59.0	57.0	5.1	51.0	57.0	0.4	51.0	59.0	0.8
aim100-1.6-4	100	160	43	48	0.1	43	48	0.1	43	48	0.3	43	48	2.5	43.0	50.0	48.0	0.7	43.0	50.0	48.0	4.6	43.0	48.0	0.3	43.0	50.0	0.7
aim100-2.0-1	100	200	18	19	0.0	18	19	0.1	18	19	0.1	18	19	0.5	18.0	20.0	19.0	0.5	18.0	20.0	19.0	2.1	18.0	19.0	0.1	18.0	20.0	0.4
aim100-2.0-2	100	200	35	39	0.0	35	39	0.1	35	39	0.2	35	39	0.9	35.0	41.0	39.0	0.7	35.0	41.0	39.0	4.9	35.0	39.0	0.3	35.0	41.0	1.4
aim100-2.0-3	100	200	25	27	0.0	25	27	0.1	25	27	0.1	25	27	1.8	25.0	30.0	27.0	0.6	25.0	30.0	27.0	2.5	25.0	27.0	0.2	25.0	30.0	0.6
aim100-2.0-4	100	200	26	31	0.0	26	31	0.1	26	31	0.2	26	32	1.6	26.0	33.0	31.0	0.6	26.0	33.0	31.0	2.2	26.0	31.0	0.2	26.0	33.0	0.6
aim200-1.6-1	200	320	52	55	0.1	52	55	0.1	52	55	0.6	52	55	2.6	52.0	55.0	55.0	1.9	52.0	55.0	55.0	8.7	52.0	55.0	0.8	52.0	55.0	5.1
aim200-1.6-2	200	320	77	80	0.1	77	80	0.2	77	80	0.9	76	82	43.0	77.0	85.0	80.0	2.5	77.0	85.0	80.0	16.1	77.0	80.0	1.7	77.0	85.0	45.5
aim200-1.6-3	200	320	77	83	0.2	77	83	0.2	77	83	1.2	77	86	300.0	77.0	89.0	83.0	2.7	77.0	89.0	83.0	11.4	77.0	83.0	1.7	77.0	89.0	432.4
aim200-1.6-4	200	320	44	46	0.1	44	46	0.1	44	46	0.6	44	46	2.3	44.0	49.0	46.0	1.7	44.0	49.0	46.0	3.4	44.0	46.0	0.5	44.0	49.0	0.9
aim200-2.0-1	200	400	49	53	0.1	49	53	0.1	49	53	0.5	49	54	3.7	49.0	56.0	53.0	2.1	49.0	56.0	53.0	9.0	49.0	53.0	0.7	49.0	56.0	45.9
aim200-2.0-2	200	400	46	50	0.1	46	50	0.1	46	50	0.7	46	50	3.0	46.0	52.0	50.0	2.2	46.0	52.0	50.0	11.5	46.0	50.0	0.7	46.0	52.0	27.5
aim200-2.0-3	200	400	35	37	0.1	35	37	0.1	35	37	0.4	35	37	0.4	35.0	38.0	37.0	1.9	35.0	38.0	37.0	6.9	35.0	37.0	0.6	35.0	38.0	10.9
aim200-2.0-4	200	400	36	42	0.1	36	42	0.1	36	42	0.6	36	42	0.8	36.0	45.0	42.0	1.7	36.0	45.0	42.0	5.7	36.0	42.0	0.6	36.0	45.0	7.5

résultats sont présentés dans le Tableau 4.9. Pour HS-V et HS-C, les valeurs données représentent la cardinalité des hitting sets trouvés par chaque algorithme. Quand la limite de temps était atteinte, les valeurs sont précédées par un "$\geq$" pour indiquer qu'elles représentent des bornes inférieures sur le nombre de variables ou clauses d'un IIS. Pour ces instances, la méthode P+Insertion a obtenu 10 réussites pour l'instance jnh2, par contre pour les autres instances il n'y avait pas 10 réussites mais toujours au moins trois réussites en mode variables et sept réussites en mode contraintes. Nous observons que pour ces grandes instances, HS-V et HS-C n'ont pas pu trouver des IIS avec un nombre minimum de clauses ou de variables à l'exception de l'instance *jnh2*. De plus, l'algorithme P+Insertion a pu extraire un IIS-V mais aucun IIS-C pour les instances *jnh16* et *jnh18* avec la limite de temps d'une heure (cpu).

Nous observons aussi que Removal et P+Insertion trouvent, dans la plupart des cas, des IIS contenant moins de variables et clauses que ceux trouvés par les autres algorithmes. Cependant, zCore+ et AMUSE+ sont significativement plus rapides. Bien qu'il soit plus lent, l'algorithme P+Insertion extrait des IIS avec beaucoup moins de variables et de clauses que Removal. Ainsi, pour l'instance *jnh13*, Removal obtient un IIS-C de 66 variables et 92 clauses (avec une moyenne de 77.3 variables et 92.0 clauses), alors que P+Insertion trouve un IIS-C de 45 variables et 55 clauses (avec une moyenne de 45.2 variables et 55.0 clauses). Ceci est dû au fait que l'heuristique basée sur le poids du voisinage guide la méthode Insertion vers des IIS plus petits. Finalement, nous pouvons voir que les bornes obtenues par les méthodes HS sont généralement loin des valeurs obtenues par les deux autres méthodes. Par contre, nous ne savons pas si ces bornes sont très proches ou très éloignées des véritables cardinalités minimum d'un IIS.

Instances SSA et BF Les instances SSA et BF ont beaucoup plus de variables et clauses que les instances AIM et JNH (quelques milliers versus quelques centaines). Le tableau 4.10 présente les résultats obtenus avec zCore+, AMUSE+, Removal, P+Insertion et l'algorithme de Mazure et al. Ces instances sont trop grandes pour

Tableau 4.9 – Résultats pour les instances JNH.

Instance			zCore+			AMUSE+			Bruni			Removal				P+Insertion				HS-V	HS-C
Nom	V	C	V	C	t	V	C	t	V	C	t	V	C	C'	t	V	C	C'	t	V	C
jnh2	100	850	61	69	0.2	48	54	0.2	51	60	3.2	63.1	181.3	45.0	1.2	44.5	76.1	45.0	54.9	41	45
jnh3	100	850	96	215	0.9	91	190	0.9	92	173	29.7	89.0	516.3	168.0	2.9	82.0	473.0	148.5	198.3	≥27	≥74
jnh4	100	850	86	128	0.3	71	102	0.3	86	140	8.2	84.6	247.9	134.0	2.3	64.0	177.0	102.0	146.4	≥29	≥60
jnh5	100	850	69	80	0.2	85	131	0.4	85	125	7.7	74.2	277.8	82.0	1.5	64.7	174.7	74.3	106.1	≥28	≥60
jnh6	100	850	95	181	0.6	81	152	0.6	88	159	22.9	79.1	332.1	130.0	2.2	73.0	263.0	127.9	145.0	≥31	≥61
jnh8	100	850	67	80	0.2	84	136	0.4	70	91	0.6	74.3	268.2	86.0	2.2	60.0	150.3	79.2	102.9	≥24	≥39
jnh9	100	850	79	104	0.2	66	87	0.2	78	118	1	80.5	350.0	114.0	1.9	57.0	126.5	71.9	114.6	≥27	≥45
jnh10	100	850	81	119	0.3	72	99	0.3	95	161	0.1	75.6	307.2	77.0	1.8	58.5	137.8	73.0	96.5	≥27	≥54
jnh11	100	850	88	140	0.3	90	168	0.6	79	129	19	81.0	356.1	116.0	2.0	70.0	211.0	110.7	263.8	≥28	≥77
jnh13	100	850	46	54	0.1	62	71	0.2	77	106	0.1	77.3	310.3	92.0	1.7	45.2	88.7	55.0	66.9	≥28	≥50
jnh14	100	850	66	78	0.2	70	92	0.2	87	124	0.5	73.1	271.0	70.0	1.5	57.0	133.8	68.0	96.2	≥26	≥48
jnh15	100	850	86	145	0.4	82	134	0.5	87	140	1.4	77.0	313.2	114.0	1.8	66.5	194.0	92.6	188.6	≥25	≥44
jnh16	100	850	100	451	10.9	99	384	10.6	100	321	55.8	93.7	634.6	281.0	7.6	92.0	600.0	-	>1*h*	≥35	≥161
jnh18	100	850	96	196	0.6	94	223	0.9	91	168	40.6	88.0	491.7	166.0	2.9	81.0	364.5	-	>1*h*	≥25	≥69
jnh19	100	850	90	143	0.4	86	153	0.5	78	122	7.4	79.3	344.0	110.0	2.0	72.0	239.0	116.0	100.9	≥27	≥50
jnh20	100	850	74	95	0.2	79	121	0.3	81	120	0.7	79.9	336.5	128.0	2.0	62.5	170.5	86.9	142.2	≥24	≥46

les méthodes `HittingSet`. À nouveau, nous observons que `P+Insertion` obtient dans de nombreux cas des IIS-C avec moins de variables et clauses que ceux obtenus par les autres algorithmes, bien qu'il soit significativement plus lent que `zCore+` et `AMUSE+`. Rappelons que tant `zCore+` que `AMUSE+` utilisent l'algorithme `zMinimal` qui est similaire à notre algorithme `Removal` pour trouver des IIS-C. La différence des résultats sur les IIS obtenus (en termes du nombre de contraintes et de variables) entre notre `Removal` et `zCore+` et `AMUSE+` provient certainement de la première partie de ces deux algorithmes, i.e., `zCore` et `AMUSE` et non de l'utilisation de `zMinimal`. Le petit nombre de variables dans les IIS-C trouvés par `Removal` peut être expliqué par sa première phase qui consiste à appliquer `Removal` pour trouver un IIS-V (duquel des clauses sont ensuite enlevées pour obtenir un IIS-C).

De plus, nous remarquons que, comme attendu, la méthode `P+Insertion` est plus rapide que `Removal` pour trouver des IIS contenant peu de variables et clauses, mais beaucoup plus lente dans les autres cas. Ainsi, pour l'instance *ssa6288-047*, `P+Insertion` trouve le même IIS-C que `Removal` dans un temps moyen de 373.9 secondes, alors que le temps moyen de `Removal` est de 1440.0 secondes. Pour ces instances, à part pour l'instance *ssa6288-047* nous n'avons pas obtenu 10 réussites sur les 10 relances de

la méthode `P+Insertion`. Bien entendu, quand le signe '-' apparaît dans une colonne c'est qu'il n'y a eu aucune réussite. Quand un résultat est donné, il y a eu au moins trois réussites.

Tableau 4.10 – Résultats pour les instances SSA et BF.

Instance			zCore+			AMUSE+			Mazure			Removal				P+Insertion			
Nom	V	C	V	C	t	V	C	t	V	C	t	V	C	C'	t	V	C	C'	t
ssa0432-003	435	1027	307	323	1.2	303	328	1.3	306	320	0.8	296.6	586.6	309.0	10.3	335.0	714.5		$->10h$
ssa2670-130	1359	3321	559	787	7.5	558	712	7.2	552	669	$2.23h$	552.4	1280.4	669.0	83.0	570.0	1346.0		$->10h$
ssa2670-141	986	2315	580	1303	17.9	580	1280	26.3	579	1247	$1.84h$	575.0	1366.0	1277.0	87.9	587.0	1386.0		$->10h$
ssa6288-047	10410	34238	24	25	1.7	24	25	2.9	-	-	$->24h$	24.0	45.0	25.0	$0.4h$	24.0	45.0	25.0	373.9
bf0432-007	1040	3668	671	1355	25.5	664	1333	46.2	674	1252	85.3	597.7	1421.3	1151.0	120.9	659.0	1616.0		$->10h$
bf1355-075	2180	6778	81	152	0.9	80	151	1.4	82	185	26.2	79.0	199.0	150.0	127.6	80.0	203.0	150.0	88.3
bf1355-638	2177	4768	82	153	0.9	81	152	1.1	83	154	32.6	81.0	203.0	152.0	126.3	82.0	207.0	153.0	128.1
bf2670-001	1393	3434	74	133	0.5	77	137	0.8	79	139	519.4	74.0	152.0	141.0	43.9	73.0	147.0	132.0	51.3

4.5.2 Instances Daimler Chrysler

Le second ensemble d'instances de test, qui peut être trouvé en [2], représente différentes propriétés de consistance de la base de données des lignes de voitures de Daimler Chrysler. Les résultats obtenus pour ces instances sont présentés dans les Tableaux 4.11, 4.12, 4.13 et 4.14. Dans le Tableau 4.11, nous montrons les résultats sur les instances testées par Huang [78] (avec `MUP`) obtenus avec `zCore+`, `AMUSE+`, `MUP`, `Removal`, `P+Insertion`, `HS-C` et `HS-V`. Nous avons imposé une limite de temps de 10 heures. Nous observons que `Removal` est beaucoup plus lent que `zCore+`, `AMUSE+` et `MUP`, tout en produisant des IIS-C ayant environ la même taille. De plus, nous voyons une fois de plus que `P+Insertion` est plus rapide que `Removal` quand les IIS sont petits, mais échoue à trouver des IIS plus grands. Finalement, nous remarquons que la méthode `HittingSet` convient mieux pour trouver des IIS de cardinalité minimum de clauses que de variables. De plus, la méthode `HS-C` trouve en une heure un IIS de cardinalité minimum pour chaque instance.

Dans le Tableau 4.12, nous comparons les résultats sur les instances testées par Mneimneh et al. [106] (pour leur algorithme SMUS) obtenus avec zCore+, AMUSE+, SMUS, Removal, HS-C et HS-V avec une limite de temps de 10 heures. Ces instances étaient trop difficiles pour l'heuristique tabou de MaxWSAT de notre méthode Insertion, c'est pour cela que nous ne reportons pas de résultats pour P+Insertion. Dans les cas où SMUS a échoué à trouver un IIS de cardinalité minimum, nous donnons à la place des bornes inférieures et supérieures, séparées par un " :", sur le nombre de clauses d'un IIS. Comme nous pouvions nous y attendre, puisque notre algorithme HS-C et SMUS garantissent de trouver des IIS de cardinalité minimum, ils trouvent effectivement des IIS-C avec le même nombre de clauses. HS-C requiert moins de temps de calcul que SMUS, même si les expérimentations faites avec SMUS ont été effectuées sur un ordinateur plus performant. De plus, bien que HS-C trouve des IIS qui ont un nombre minimum de clauses, il est plus rapide que Removal. Ceci est dû au fait que, contrairement à l'algorithme Removal, il construit les IIS avec une approche constructive, et en plus les IIS de cardinalité minimum ont beaucoup moins de clauses que les instances originales. À nouveau, zCore+ et AMUSE+ sont les méthodes les plus rapides.

Dans les Tableaux 4.13 et 4.14, nous comparons les résultats obtenus sur toutes les instances Daimler Chrysler avec zCore+, AMUSE+, CAMUS et HS-C. Quand CAMUS réussit à trouver tous les IIS d'une instance avec la limite de temps CPU de 600 secondes, "#IIS" donne le nombre d'IIS différents contenus dans cette instance, alors que "Taille IIS" donne le plus petit et plus grand nombre de clauses de ces IIS. Sinon, si "#IIS" est précédé par ">", les valeurs données sont celles calculées en utilisant les IIS trouvés avant l'arrêt. Finalement, si "-" est donné pour toutes ces valeurs, alors aucun IIS n'a été trouvé par leur algorithme dans la limite de temps imposée. Nous voyons que, alors que CAMUS a été capable de déterminer le plus petit nombre de clauses d'un IIS pour 32 des 84 instances, avec la limite de temps de 600 secondes, notre méthode HS-C a trouvé cette valeur pour 79 instances avec la même limite de temps. À nouveau, zCore+ et

AMUSE+ sont les algorithmes les plus rapides, mais ils n'exhibent pas forcément un IIS-C avec le plus petit nombre de clauses. Le nombre de réussites pour la méthode P+Insertion a toujours été de 10 (sur 10) pour le mode contraintes. En mode variables, l'instance *168_FW_UT_2468* a obtenu 4 réussites et l'instance *202_FS_RZ_44* a obtenu 6 réussites et toutes les autres instances 10 réussites.

Tableau 4.11 – Quelques résultats pour les instances Daimler Chrysler.

Instance			zCore+			AMUSE+			MUP			Removal				P+Insertion				HS-C			HS-V		
Nom	V	C	V	C	t	V	C	t	V	C	t	V	C	C'	t	V	C	C'	t	V	C	t	V	C	t
168_FW_SZ_107	1698	6599	41	47	0.8	43	50	1.1	41	47	0.1	41.0	206.0	48.0	98.6	41.6	215.5	59.0	78.8	41.0	47.0	32.5	41.0	208.2	69.1
168_FW_UT_2468	1909	7487	32	35	0.5	32	35	0.8	32	35	0.1	30.0	125.0	33.0	102.8	31.3	129.3	47.0	28.4	30.0	33.0	40.6	30.0	125.0	75.3
202_FS_RZ_44	1750	6199	12	18	0.3	21	27	0.5	12	18	0.1	17.0	141.0	32.0	72.4	12.0	59.0	28.7	7.3	12.1	18.0	11.9	12.0	59.0	11.6
202_FS_SZ_84	1750	6273	204	219	1.2	205	220	1.2	206	221	0.3	199.0	1923.0	218.0	213.9	-	-	-	$>10h$	199.0	214.0	599.3	198.0	1929.0	$0.8h$
202_FS_SZ_97	1750	6250	26	33	0.4	24	34	0.6	26	33	0.1	21.0	97.0	28.0	83.8	21.0	97.0	32.6	14.2	21.0	28.0	17.1	21.0	97.0	25.5
202_FW_SZ_100	1799	8738	22	26	0.4	29	30	0.8	24	28	0.1	21.0	106.0	29.0	106.2	21.5	109.4	30.7	18.1	22.0	23.0	16.7	21.0	106.0	19.9
202_FW_SZ_103	1799	10283	139	158	2.5	138	156	3.4	140	159	0.5	134.0	1641.0	156.0	$0.3h$	133.0	1757.0	334.0	$6.3h$	133.0	148.0	424.5	≥ 28	-	$>10h$
202_FW_SZ_87	1799	8946	243	379	3.1	244	398	3.6	247	383	0.6	234.0	2831.0	382.0	372.8	-	-	-	$>10h$	231.0	361.0	$1.0h$	231.0	2828.0	$0.8h$
202_FW_SZ_96	1799	8849	205	210	1.5	215	220	1.8	210	215	0.3	205.0	2377.0	210.0	286.5	-	-	-	$>10h$	204.0	209.0	983.8	204.0	2392.0	$0.9h$
210_FS_SZ_55	1755	5781	28	45	0.3	27	46	0.6	29	46	0.1	24.0	109.0	42.0	70.4	24.1	111.4	62.0	10.5	24.0	41.0	31.4	24.0	109.0	29.1
210_FW_SZ_90	1789	7994	216	279	3.1	228	291	3.9	221	284	0.5	209.0	2044.0	287.0	408.5	-	-	-	$>10h$	209.0	274.0	$0.34h$	≥ 64	-	$>10h$
210_FW_SZ_91	1789	7721	214	277	2.6	216	279	3.1	225	288	0.5	206.0	2052.0	280.1	362.8	240.0	2593.0	-	$>10h$	206.0	271.0	$0.3h$	≥ 28	-	$>10h$
210_FW_UT_8630	2024	9721	29	38	0.6	25	38	1.1	23	35	0.1	19.0	88.0	31.0	139.5	19.0	88.0	30.0	16.9	19.0	30.0	39.7	19.0	88.0	67.8
220_FV_SZ_55	1728	5753	240	304	3.8	238	305	4.6	246	310	0.6	229.0	1452.0	356.0	427.6	229.0	1452.0	499.0	$5.0h$	233.0	297.0	664.9	≥ 40	-	$>10h$
220_FV_SZ_65	1728	4496	18	23	0.2	18	23	0.4	24	29	0.1	18.0	50.0	23.0	54.9	18.0	50.0	36.0	5.5	18.3	23.0	9.7	18.0	50.0	9.6

Tableau 4.12 – D'autres résultats pour les instances Daimler Chrysler.

Instance			zCore+			AMUSE+			SMUS		Removal				HS-C			HS-V		
Nom	V	C	V	C	t	V	C	t	C	t	V	C	C'	t	V	C	t	V	C	t
168_FW_SZ_107	1698	6599	41	47	0.8	43	50	1.1	47	546.5	41.0	206.0	48.0	98.6	41.0	47.0	32.5	41.0	208.2	69.1
168_FW_SZ_41	1698	5387	27	30	0.3	25	27	0.5	26	257.4	23.0	86.0	27.0	60.9	24.9	26.0	37.0	23.0	86.0	93.3
168_FW_SZ_66	1698	5401	12	16	0.2	12	16	0.5	16	18.8	12.0	44.0	16.0	65.8	12.0	16.0	11.6	12.0	44.0	6.3
168_FW_UT_2463	1909	7489	35	38	0.5	35	38	0.7	35	350.4	32.0	165.0	37.2	105.0	32.0	35.0	50.6	32.0	154.8	112.0
168_FW_UT_2469	1909	7500	30	33	0.4	30	33	0.7	32	831.5	29.0	102.0	32.0	93.4	29.0	32.0	38.8	29.0	119.2	133.7
168_FW_UT_714	1909	7487	6	9	0.4	6	9	0.7	9	14.5	6.0	17.0	10.0	83.3	6.0	9.0	4.0	6.0	17.0	6.4
168_FW_UT_851	1909	7491	9	10	0.4	9	10	0.7	8	59.9	7.0	27.0	8.0	83.5	7.0	8.0	5.5	7.0	27.0	9.6
170_FR_RZ_32	1659	4956	30	327	0.8	30	327	0.7	227	121.3	30.0	870.0	227.0	75.3	30.0	227.0	90.4	30.0	870.0	16.4
170_FR_SZ_58	1659	5001	47	49	0.3	47	49	0.4	46	15.3	43.0	176.0	47.0	56.5	43.0	46.0	19.7	43.0	176.0	58.1
170_FR_SZ_92	1659	5082	46	131	0.5	46	131	0.6	131	15.1	46.0	137.0	131.0	51.6	46.0	131.0	39.7	46.0	137.0	26.5
170_FR_SZ_96	1659	4955	22	64	0.3	22	63	0.4	53	322.8	21.0	301.0	62.0	59.5	22.0	53.0	28.6	21.0	301.0	11.8
202_FS_RZ_44	1750	6199	12	18	0.3	21	27	0.5	18	131.0	17.0	141.0	32.0	72.4	12.1	18.0	11.9	12.0	59.0	11.6
202_FS_SZ_104	1750	6201	30	35	0.3	29	34	0.5	24	5.0	19.0	116.0	24.0	76.2	19.0	24.0	15.3	19.0	116.0	26.6
202_FS_SZ_121	1750	6181	20	22	0.3	21	23	0.5	22	2.5	20.0	38.0	22.0	64.9	20.0	22.0	11.4	20.0	38.0	19.9
202_FS_SZ_122	1750	6179	19	33	0.3	19	33	0.6	33	3.8	19.0	216.0	33.0	76.4	19.0	33.0	10.2	19.0	216.0	15.1
202_FS_SZ_74	1750	6355	34	150	0.7	34	150	0.7	150	36.4	34.0	234.0	150.0	79.5	34.0	150.0	88.0	34.0	236.0	103.3
202_FS_SZ_84	1750	6273	204	219	1.2	205	220	1.2	39 :216	$1.1h$	199.0	1923.0	218.0	213.9	199.0	214.0	599.3	198.0	1929.0	$0.8h$
202_FS_SZ_97	1750	6250	26	33	0.4	24	34	0.6	28	62.1	21.0	97.0	28.0	83.9	21.0	28.0	17.1	21.0	97.0	25.5
202_FW_RZ_57	1799	8685	30	213	1.0	30	213	1.3	213	58.3	30.0	815.0	213.0	113.4	30.0	213.0	81.3	30.0	815.0	42.3
202_FW_SZ_100	1799	8738	22	26	0.4	29	30	0.8	23	174.0	21.0	106.0	29.0	106.2	22.0	23.0	16.7	21.0	106.0	19.9
202_FW_SZ_103	1799	10283	139	158	2.5	138	156	3.4	35 :149	$2.4h$	134.0	1641.0	156.0	$0.3h$	133.0	148.0	424.5	≥ 28	-	$>10h$
202_FW_SZ_123	1799	8686	21	37	0.5	21	37	0.7	36	14.7	20.0	248.0	36.0	100.2	20.0	36.0	13.2	20.0	248.0	15.3
202_FW_SZ_61	1799	8745	16	18	0.4	16	18	0.7	18	163.8	15.0	67.0	19.0	98.5	16.0	18.0	52.7	15.0	67.0	68.3
202_FW_SZ_77	1799	8860	34	156	0.8	34	156	1.0	156	37.2	34.0	240.0	156.0	103.8	34.0	156.0	136.0	34.0	240.0	100.5
202_FW_SZ_98	1799	8689	6	8	0.4	16	21	0.7	7	58.2	6.0	20.0	7.1	89.0	6.0	7.0	6.5	6.0	20.0	7.0
208_FA_RZ_43	1608	5297	8	9	0.2	6	8	0.4	8	76.9	7.0	19.0	9.0	54.2	6.0	8.0	9.2	6.0	19.0	3.9
208_FA_SZ_120	1608	5278	19	34	0.2	19	34	0.4	34	3.9	19.0	247.0	34.0	59.0	19.0	34.0	9.6	19.0	247.0	9.2
208_FA_SZ_87	1608	5299	17	20	0.2	17	19	0.4	18	15.1	17.0	136.0	21.0	54.7	17.0	18.0	15.1	17.0	136.0	12.7
208_FA_UT_3254	1876	7334	40	68	0.5	41	70	0.8	40	95.3	38.0	883.0	40.0	109.9	38.0	40.0	18.6	38.0	883.0	35.2
208_FA_UT_3255	1876	7337	40	68	0.5	41	70	0.8	40	94.6	38.0	883.0	40.0	109.8	38.0	40.0	18.7	38.0	883.0	38.0
210_FS_RZ_23	1755	5778	22	35	0.3	23	41	0.6	31	266.3	17.0	71.0	33.0	68.6	17.0	31.0	24.2	17.0	71.0	24.1
210_FS_RZ_38	1775	5763	17	28	0.3	19	34	0.5	25	261.9	13.0	57.0	25.0	66.3	13.0	25.0	15.9	13.0	57.0	18.0
210_FS_RZ_40	1775	5752	29	140	0.6	29	140	0.6	140	36.2	29.0	652.0	140.0	81.5	29.0	140.0	47.9	29.0	652.0	41.9
210_FS_SZ_103	1775	5775	35	45	0.3	39	50	0.6	45	386.4	35.0	144.0	45.0	71.1	35.0	45.0	31.9	35.0	144.0	513.5
210_FS_SZ_107	1775	5762	11	15	0.3	11	15	0.5	15	25.3	11.0	35.0	15.0	65.0	11.0	15.0	6.9	11.0	35.0	11.5
210_FS_SZ_123	1775	5921	49	176	0.7	49	176	0.8	176	$0.4h$	49.0	318.0	176.0	75.5	49.0	176.0	83.9	49.0	318.0	347.5
210_FS_SZ_78	1775	5930	37	170	0.7	37	170	0.8	170	56.7	34.0	239.0	180.0	74.8	37.0	170.0	116.5	34.0	239.0	114.9
210_FW_RZ_57	1789	7405	14	26	0.4	17	32	0.7	25	355.0	13.0	57.0	25.0	93.1	13.0	25.0	17.3	13.0	57.0	27.7
210_FW_RZ_59	1789	7394	29	140	0.6	29	140	0.8	140	56.7	29.0	652.0	140.0	107.7	29.0	140.0	51.3	29.0	652.0	96.7
210_FW_SZ_106	1789	7417	39	55	0.4	42	53	0.8	49	789.6	38.0	205.0	51.0	97.6	38.0	49.0	37.2	37.0	201.0	741.6
210_FW_SZ_111	1789	7404	11	15	0.4	11	15	0.6	15	35.2	11.0	36.0	15.0	99.5	11.0	15.0	7.8	11.0	36.0	29.2
210_FW_SZ_128	1789	7412	14	23	0.4	13	23	0.7	22	151.3	12.0	30.0	23.0	97.3	13.0	22.0	20.0	12.0	30.0	35.0
210_FW_SZ_90	1789	7994	216	279	3.1	228	291	3.9	37 :275	$1.8h$	209.0	2044.0	287.0	408.5	209.0	274.0	$0.34h$	≥ 64	-	$>10h$
210_FW_SZ_91	1789	7721	214	277	2.6	216	279	3.1	41 :271	$1.8h$	206.0	2052.0	280.1	362.7	206.0	271.0	$0.3h$	≥ 28	-	$>10h$
220_FV_RZ_12	1728	4512	10	11	0.2	13	14	0.4	11	50.6	16.0	81.0	17.0	50.6	10.0	11.0	5.6	10.0	40.0	11.1
220_FV_RZ_13	1728	4509	9	10	0.2	13	14	0.3	10	33.4	13.0	42.0	14.0	49.2	9.0	10.0	5.6	9.0	31.0	8.0
220_FV_RZ_14	1728	4508	10	11	0.2	10	11	0.4	11	33.9	10.0	40.0	11.0	53.7	10.0	11.0	3.8	10.0	40.0	7.3
220_FV_SZ_46	1728	4498	16	17	0.2	31	32	0.4	17	78.4	16.0	78.0	17.0	50.4	16.0	17.0	10.7	16.0	78.0	31.8
220_FV_SZ_65	1728	4496	18	23	0.2	18	23	0.4	23	18.9	18.0	50.0	23.0	54.9	18.3	23.0	9.7	18.0	50.0	16.4

Tableau 4.13 – D'autres résultats pour les instances Daimler Chrysler.

Instance			zCore+			AMUSE+			CAMUS				HS-C		
Nom	V	C	V	C	t	V	C	t	t	#IIS	Taille IIS Min	Max	V	C	t
168_FW_SZ_107	1698	6599	41	47	0.8	43	50	1.1	-	-	-	-	41.0	47.0	32.5
168_FW_SZ_128	1698	5425	30	96	0.6	29	96	0.7	-	-	-	-	28.0	92.0	52.9
168_FW_SZ_41	1698	5387	27	30	0.3	25	27	0.5	-	-	-	-	24.9	26.0	37.0
168_FW_SZ_66	1698	5401	12	16	0.2	12	16	0.5	-	-	-	-	12.0	16.0	11.6
168_FW_SZ_75	1698	5422	47	48	0.3	47	48	0.5	-	-	-	-	47.0	48.0	29.1
168_FW_UT_2463	1909	7489	35	38	0.5	35	38	0.7	-	-	-	-	32.0	35.0	50.7
168_FW_UT_2468	1909	7487	32	35	0.5	32	35	0.8	-	-	-	-	30.0	33.0	40.6
168_FW_UT_2469	1909	7500	30	33	0.4	30	33	0.7	-	-	-	-	29.0	32.0	38.8
168_FW_UT_714	1909	7487	6	9	0.4	6	9	0.7	-	-	-	-	6.0	9.0	4.0
168_FW_UT_851	1909	7491	9	10	0.4	9	10	0.7	0.3	102	8	16	7.0	8.0	5.5
168_FW_UT_852	1909	7489	9	10	0.4	9	10	0.7	0.3	102	8	16	7.0	8.0	5.4
168_FW_UT_854	1909	7486	9	10	0.4	9	10	0.7	0.3	102	8	16	7.0	8.0	5.1
168_FW_UT_855	1909	7485	9	10	0.4	9	10	0.7	0.3	102	8	16	7.0	8.0	5.2
170_FR_RZ_32	1659	4956	30	327	0.8	30	327	0.7	0.8	32768	227	228	30.0	227.0	90.4
170_FR_SZ_58	1659	5001	47	49	0.3	47	49	0.4	7.5	218692	46	63	43.0	46.0	19.7
170_FR_SZ_92	1659	5082	46	131	0.5	46	131	0.6	0.1	1	131	131	46.0	131.0	39.7
170_FR_SZ_95	1659	4955	22	63	0.3	21	62	0.5	-	>23301932	53	66	22.0	52.0	27.0
170_FR_SZ_96	1659	4955	22	64	0.3	22	63	0.4	-	>10383703	67	82	22.0	53.0	28.6
202_FS_RZ_44	1750	6199	12	18	0.3	21	27	0.5	-	>7764186	29	54	12.1	18.0	11.9
202_FS_SZ_104	1750	6201	30	35	0.3	29	34	0.5	-	>4803992	70	94	19.0	24.0	15.3
202_FS_SZ_121	1750	6181	20	22	0.3	21	23	0.5	0.1	4	22	24	20.0	22.0	11.4
202_FS_SZ_122	1750	6179	19	33	0.3	19	33	0.6	0.1	1	33	33	19.0	33.0	10.2
202_FS_SZ_74	1750	6355	34	150	0.7	34	150	0.7	-	-	-	-	34.0	150.0	88.0
202_FS_SZ_84	1750	6273	204	219	1.2	205	220	1.2	-	-	-	-	199.0	214.0	599.3
202_FS_SZ_95	1750	6184	11	13	0.3	13	15	0.5	-	>6173760	35	50	6.0	7.0	5.4
202_FS_SZ_97	1750	6250	26	33	0.4	24	34	0.6	-	>5466033	99	125	21.0	28.0	17.1
202_FW_RZ_57	1799	8685	30	213	1.0	30	213	1.3	0.4	1	213	213	30.0	213.0	81.3
202_FW_SZ_100	1799	8738	22	26	0.4	29	30	0.8	-	-	-	-	22.0	23.0	16.7
202_FW_SZ_103	1799	10283	139	158	2.5	138	156	3.4	-	-	-	-	133.0	148.0	424.5
202_FW_SZ_118	1799	8811	47	130	0.8	47	130	0.9	-	>11072627	130	132	46.0	129.0	110.5
202_FW_SZ_123	1799	8686	21	37	0.5	21	37	0.7	0.2	4	36	38	20.0	36.0	13.2
202_FW_SZ_124	1799	8684	19	33	0.4	19	33	0.8	0.1	1	33	33	19.0	33.0	11.2
202_FW_SZ_61	1799	8745	16	18	0.4	16	18	0.7	-	-	-	-	16.0	18.0	52.7
202_FW_SZ_77	1799	8860	34	156	0.8	34	156	1.0	-	-	-	-	34.0	156.0	135.9
202_FW_SZ_87	1799	8946	243	379	3.1	244	398	3.6	-	-	-	-	231.0	361.0	3519.9
202_FW_SZ_96	1799	8849	205	210	1.5	215	220	1.8	-	-	-	-	204.0	209.0	983.8
202_FW_SZ_98	1799	8689	6	8	0.4	16	21	0.7	-	-	-	-	6.0	7.0	6.5
202_FW_UT_2814	2038	11352	22	28	0.6	15	19	1.1	-	-	-	-	15.0	16.0	95.1
202_FW_UT_2815	2038	11352	22	28	0.6	15	19	1.1	-	-	-	-	15.0	16.0	96.9

Tableau 4.14 – D'autres résultats pour les instances Daimler Chrysler.

Instance			zCore+			AMUSE+			CAMUS				HS-C		
Nom	V	C	V	C	t	V	C	t	t	#IIS	Taille IIS Min	Max	V	C	t
208_FA_RZ_43	1608	5297	8	9	0.2	6	8	0.4	-	>87515	9	28	6.0	8.0	9.2
208_FA_RZ_64	1608	5279	29	212	0.7	29	212	0.7	0.2	1	212	212	29.0	212.0	67.0
208_FA_SZ_120	1608	5278	19	34	0.2	19	34	0.4	0.1	2	34	34	19.0	34.0	9.6
208_FA_SZ_121	1608	5278	18	32	0.2	18	32	0.4	0.1	2	32	32	18.0	32.0	9.1
208_FA_SZ_87	1608	5299	17	20	0.2	17	19	0.4	0.9	12884	18	27	17.0	18.0	15.1
208_FA_UT_3254	1876	7334	40	68	0.5	41	70	0.8	0.7	17408	40	74	38.0	40.0	18.6
208_FA_UT_3255	1876	7337	40	68	0.5	41	70	0.8	1.4	52736	40	74	38.0	40.0	18.7
208_FC_RZ_65	1654	5591	12	14	0.2	11	12	0.4	-	-	-	-	11.0	12.0	13.8
208_FC_RZ_70	1654	5543	29	212	0.8	29	212	0.7	0.2	1	212	212	29.0	212.0	67.5
208_FC_SZ_107	1654	5641	36	47	0.3	38	50	0.5	-	-	-	-	31.0	44.0	26.1
208_FC_SZ_127	1654	5542	19	34	0.2	19	34	0.5	0.1	1	34	34	19.0	34.0	10.2
208_FC_SZ_128	1654	5542	18	32	0.3	18	32	0.5	0.1	1	32	32	18.0	32.0	9.6
210_FS_RZ_23	1755	5778	22	35	0.3	23	41	0.6	-	-	-	-	17.0	31.0	24.2
210_FS_RZ_38	1755	5763	17	28	0.3	19	34	0.5	-	>5365108	46	59	13.0	25.0	15.9
210_FS_RZ_40	1755	5752	29	140	0.6	29	140	0.6	0.3	15	140	173	29.0	140.0	47.9
210_FS_SZ_103	1755	5755	35	45	0.3	39	50	0.6	-	-	-	-	35.0	45.0	31.9
210_FS_SZ_107	1755	5762	11	15	0.3	11	15	0.5	-	>2160988	48	65	11.0	15.0	6.9
210_FS_SZ_123	1755	5921	49	176	0.7	49	176	0.8	-	>10064216	177	234	49.0	176.0	83.9
210_FS_SZ_129	1755	5753	20	33	0.3	20	33	0.5	0.1	1	33	33	20.0	33.0	10.2
210_FS_SZ_130	1755	5753	19	31	0.3	19	31	0.5	0.1	1	31	31	19.0	31.0	9.6
210_FS_SZ_55	1755	5781	28	45	0.3	27	46	0.6	-	-	-	-	24.0	41.0	31.4
210_FS_SZ_78	1755	5930	37	170	0.7	37	170	0.8	-	-	-	-	37.0	170.0	116.5
210_FW_RZ_30	1789	7426	22	39	0.4	27	48	0.8	-	-	-	-	19.0	35.0	36.4
210_FW_RZ_57	1789	7405	14	26	0.4	17	32	0.7	-	>4597505	47	61	13.0	25.0	17.3
210_FW_RZ_59	1789	7394	29	140	0.6	29	140	0.8	0.4	15	140	173	29.0	140.0	51.3
210_FW_SZ_106	1789	7417	39	55	0.4	42	53	0.8	-	-	-	-	38.0	49.0	37.2
210_FW_SZ_111	1789	7404	11	15	0.4	11	15	0.6	-	>6904528	39	51	11.0	15.0	7.8
210_FW_SZ_128	1789	7412	14	23	0.4	13	23	0.7	-	-	-	-	13.0	22.0	20.0
210_FW_SZ_129	1789	7606	49	176	0.9	49	176	1.0	-	>7528982	238	281	49.0	176.0	90.0
210_FW_SZ_135	1789	7395	20	33	0.4	20	33	0.7	0.1	1	33	33	20.0	33.0	11.6
210_FW_SZ_136	1789	7395	19	31	0.4	19	31	0.7	0.2	1	31	31	19.0	31.0	10.8
210_FW_SZ_80	1789	7572	39	173	0.8	35	175	1.0	-	-	-	-	37.0	171.0	155.3
210_FW_SZ_90	1789	7994	216	279	3.1	228	291	3.9	-	-	-	-	209.0	274.0	1225.7
210_FW_SZ_91	1789	7721	214	277	2.6	216	279	3.1	-	-	-	-	206.0	271.0	1093.4
210_FW_UT_8630	2024	9721	29	38	0.6	25	38	1.1	-	-	-	-	19.0	30 .0	39.7
210_FW_UT_8634	2024	9719	28	32	0.5	28	35	1.0	-	-	-	-	18.0	23.0	30.0
220_FV_RZ_12	1728	4512	10	11	0.2	13	14	0.4	1.7	80272	11	35	10.0	11.0	5.6
220_FV_RZ_13	1728	4509	9	10	0.2	13	14	0.3	0.3	6772	10	27	9.0	10.0	5.6
220_FV_RZ_14	1728	4508	10	11	0.2	10	11	0.4	0.1	80	11	18	10.0	11.0	3.8
220_FV_SZ_114	1728	4777	47	132	0.5	47	132	0.6	-	>1356651	320	364	47.0	132.0	62.3
220_FV_SZ_121	1728	4508	14	58	0.3	14	58	0.4	0.2	9	58	65	14.0	58 .0	17.3
220_FV_SZ_39	1728	5263	199	205	2.2	211	217	2.8	-	-	-	-	199.0	205.0	340.2
220_FV_SZ_46	1728	4498	16	17	0.2	31	32	0.4	-	>11089464	40	71	16.0	17.0	10.7
220_FV_SZ_55	1728	5753	240	304	3.8	238	305	4.6	-	>2822517	663	670	233.0	297.0	664.9
220_FV_SZ_65	1728	4496	18	23	0.2	18	23	0.4	3.2	103442	23	40	18.0	23.0	9.7

4.5.3 Instances difficiles provenant de la coloration de graphes

Les instances considérées jusqu'à maintenant sont relativement faciles à résoudre pour les algorithmes exacts (comme `zChaff`), ce qui permet de trouver facilement des IIS tant avec notre algorithme `Removal` qu'avec les techniques comme `zCore+` et `AMUSE+`. Dans le but de tester nos algorithmes dans un contexte différent, nous avons généré des instances qui sont beaucoup plus difficiles pour les algorithmes exacts. Ces cinq nouvelles instances ont été générées à partir de problèmes aléatoires de k-coloration de graphe en utilisant le convertisseur de Culberson [35]. Cette conversion d'un problème de k-coloration de graphe en un problème SAT fonctionne de la façon suivante. Supposons que nous ayons un problème de k-coloration pour un graphe $G = (V, E)$ tel que $|V| = n$ et $|E| = m$. Pour transformer ce problème de k-coloration en un problème SAT, pour chaque sommet $v \in V$, nous créons k variables : $x_{v1}, \ldots, x_{vk}$. Si x_{vi} est vraie alors v obtient la couleur i. Ensuite, nous ajoutons les clauses suivantes.

- Pour chaque sommet $v \in V$, il y a une clause $x_{v1} \vee \ldots \vee x_{vk}$. Ces contraintes imposent qu'au moins une couleur soit affectée pour chaque sommet v.
- Pour chaque arête $(v, w) \in E$ et pour chaque couleur $i = 1, \ldots, k$, il y a une clause $\bar{x}_{vi} \vee \bar{x}_{wi}$. Ces contraintes imposent qu'une couleur i ne peut pas être affectée en même temps aux deux extrémités d'une arête.

Le problème SAT CNF correspondant comporte donc kn variables et $n + mk$ clauses. Prenons pour exemple le graphe de la Figure 4.18 et intéressons-nous au problème de 2-coloration de ce graphe.

La transformation de ce problème de 2-coloration pour ce graphe en un problème SAT CNF est la suivante. Les 10 variables sont $x_{a1}, x_{a2}, x_{b1}, x_{b2}, x_{c1}, x_{c2}, x_{d1}, x_{d2}, x_{e1}, x_{e2}$, et

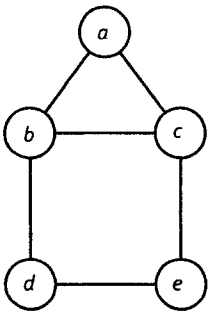


Figure 4.18 – Exemple illustrant la transformation d'un problème de 2-coloration de graphe en problème SAT CNF.

les 17 clauses sont

$$
\begin{aligned}
C_1 &= x_{a1} \vee x_{a2} \\
C_2 &= x_{b1} \vee x_{b2} \\
C_3 &= x_{c1} \vee x_{c2} \\
C_4 &= x_{d1} \vee x_{d2} \\
C_5 &= x_{e1} \vee x_{e2} \\
C_6 &= \bar{x}_{a1} \vee \bar{x}_{b1} \\
C_7 &= \bar{x}_{a2} \vee \bar{x}_{b2} \\
C_8 &= \bar{x}_{a1} \vee \bar{x}_{c1} \\
C_9 &= \bar{x}_{a2} \vee \bar{x}_{c2}
\end{aligned}
\qquad
\begin{aligned}
C_{10} &= \bar{x}_{b1} \vee \bar{x}_{c1} \\
C_{11} &= \bar{x}_{b2} \vee \bar{x}_{c2} \\
C_{12} &= \bar{x}_{b1} \vee \bar{x}_{d1} \\
C_{13} &= \bar{x}_{b2} \vee \bar{x}_{d2} \\
C_{14} &= \bar{x}_{c1} \vee \bar{x}_{e1} \\
C_{15} &= \bar{x}_{c2} \vee \bar{x}_{e2} \\
C_{16} &= \bar{x}_{d1} \vee \bar{x}_{e1} \\
C_{17} &= \bar{x}_{d2} \vee \bar{x}_{e2}
\end{aligned}
$$

Nous nous sommes demandés si les sous-ensembles incohérents obtenus correspondaient toujours à un sous-graphe du graphe original. Or, dans l'exemple présenté ci-dessus il est possible que l'IIS-C obtenu par nos algorithmes ne corresponde pas à un sous-graphe du graphe original. Par exemple, si nous utilisons `P+Insertion` ou `HS-C` nous obtenons toujours l'IIS-C correspondant au triangle. Par contre, si nous utilisons `Removal` ou `Insertion` sans heuristique du poids du voisinage nous pouvons obtenir l'IIS-C suivant qui comporte 10 variables et 13 contraintes.

$$
\begin{aligned}
C_1 &= x_{a1} \vee x_{a2} \\
C_2 &= x_{b1} \vee x_{b2} \\
C_3 &= x_{c1} \vee x_{c2} \\
C_4 &= x_{d1} \vee x_{d2} \\
C_5 &= x_{e1} \vee x_{e2} \\
C_6 &= \bar{x}_{a1} \vee \bar{x}_{b1} \\
C_7 &= \bar{x}_{a2} \vee \bar{x}_{b2} \\
C_8 &= \bar{x}_{a1} \vee \bar{x}_{c1} \\
C_9 &= \bar{x}_{a2} \vee \bar{x}_{c2}
\end{aligned}
\qquad
\begin{aligned}
C_{11} &= \bar{x}_{b2} \vee \bar{x}_{c2} \\
C_{12} &= \bar{x}_{b1} \vee \bar{x}_{d1} \\
C_{14} &= \bar{x}_{c1} \vee \bar{x}_{e1} \\
C_{17} &= \bar{x}_{d2} \vee \bar{x}_{e2}
\end{aligned}
$$

Cet IIS peut facilement se vérifier si nous réfléchissons en termes de couleurs et regardons les contraintes de l'IIS. Les contraintes C_1 à C_5 imposent que chaque sommet doit avoir une couleur. Les contraintes C_6 et C_7 imposent que a et b doivent obtenir des couleurs différentes et les contraintes C_8 et C_9 imposent que a et c doivent obtenir des couleurs différentes. Si a a la couleur 1, par la clause C_{11} b et c ne peuvent pas avoir tous les deux la couleur 2. Si a a la couleur 2, alors nous pouvons donner la couleur 1 à b et c, mais alors par les clauses C_{12} et C_{14} il faudrait donner la couleur 2 à d et e, or ceci est impossible à cause de la clause C_{17}.

En utilisant cette conversion, nous avons généré cinq instances difficiles. Pour chacune de ces instances, nous trouvons un IIS-V potentiel en utilisant la méthode `Insertion`. Ensuite, nous déterminons si ce sous-ensemble est réalisable en utilisant `zChaff`. Si ce sous-ensemble est non réalisable, nous avons une preuve que l'instance originale était non réalisable. Le Tableau 4.15 résume les résultats de ces expérimentations. Les colonnes étiquetées "zChaff" et "Vérif" donnent les temps d'exécution requis par `zChaff` pour résoudre les instances originales et les sous-formules extraites, avec une limite de temps de 24 heures. Nous pouvons observer que parmi les cinq instances testées, `zChaff` n'a pu résoudre qu'une seule instance originale dans la limite de temps, alors

qu'il a réussi à résoudre trois des sous-formules extraites. De plus, nous remarquons que `zChaff` a requis beaucoup moins de temps pour résoudre les sous-formules extraites. Par exemple, alors qu'il a échoué à résoudre l'instance *DGHP50_5* en 24 heures, il a été capable de résoudre sa sous-formule extraite en 7.74 heures. Étant donné que cela prend 0.87 heures pour extraire la sous-formule non réalisable, le temps total nécessaire pour prouver que cette instance est non réalisable est 8.61 heures. Notons que `zCore+` et `AMUSE+` étaient incapables de traiter ces instances avec la limite de temps de 24 heures. Comme ces instances sont difficiles, le nombre de réussites (parmi les dix relances) est faible : seulement une réussite pour les instances *DGHP50_1* à *DGHP50_4* et 3 réussites pour l'instance *DGHP50_5*.

Tableau 4.15 – Résultats pour des instances provenant de la coloration de graphes.

Instance			**zChaff**	**P+Insertion**			**Vérif**
Nom	**V**	**C**	**t**	**V**	**C**	**t**	**t**
DGHP50_1	350	3641	5.23h	259	2277	0.43h	2.48h
DGHP50_2	400	4618	>24h	*336	*3450	1.43h	>24h
DGHP50_3	400	4650	>24h	288	2796	0.70h	17.42h
DGHP50_4	400	4602	>24h	*288	*2724	2.22h	>24h
DGHP50_5	400	4706	>24h	280	2619	0.87h	7.74h

4.5.4 Discussion générale des résultats

Dans la section précédente, nous avons clairement observé que les algorithmes proposés sont plus lents que d'autres algorithmes tels que `zCore+` et `AMUSE+` pour extraire des IIS-C. Ils ont, cependant, plusieurs avantages quand ils sont comparés avec les algorithmes les plus populaires. Premièrement, nos algorithmes peuvent trouver non seulement des IIS-C mais aussi des IIS-V. Ainsi, par exemple, alors que la version contraintes de `Removal` est similaire à `zMinimal`, nous avons montré que la version variables de `Removal` peut d'abord être employée afin de réduire la taille d'une instance, et ensuite un IIS-C peut être extrait en utilisant la version IIS-C de `Removal`. L'algorithme ré-

sultant, appelé Removal, a parfois extrait des IIS-C avec beaucoup moins de variables et clauses que les autres algorithmes. L'algorithme P+Insertion non seulement suit la même approche (i.e., enlevant des clauses d'un IIS-V afin d'obtenir un IIS-C), mais aussi utilise une heuristique basée sur le poids du voisinage dans le but d'extraire des petits IIS. Nous avons observé qu'une telle technique rend possible, dans plusieurs cas, le fait d'obtenir des IIS-C avec significativement moins de clauses.

Il y a d'autres différences importantes entre les algorithmes Insertion et Removal. Par exemple, l'algorithme Insertion utilise une heuristique de recherche tabou pour résoudre le problème Max-SAT pondéré, alors que Removal utilise l'algorithme exact zChaff pour résoudre le problème SAT. Dans plusieurs cas, l'algorithme de recherche tabou était plus lent que zChaff, expliquant pourquoi l'algorithme Insertion était souvent plus lent que Removal. Cependant, nous avons construit des instances difficiles de coloration, qui sont très difficiles à résoudre pour zChaff, mais encore résolvables pour notre heuristique tabou. Pour ces instances, l'algorithme Insertion était le seul capable de trouver des IIS-V pour prouver la non réalisabilité de ces instances. Une autre différence est que l'algorithme Insertion construit des IIS d'une façon constructive. Donc, quand l'heuristique de recherche tabou ne fait aucune erreur, le nombre d'étapes est plus petit ou égal au nombre de clauses de l'IIS-C. Ainsi, il n'est pas surprenant que l'algorithme Insertion soit parfois plus rapide que l'algorithme Removal quand les IIS-C ont beaucoup moins de clauses que l'instance originale.

La version contraintes de l'algorithme HittingSet (i.e., HS-C) extrait des IIS-C avec un nombre minimum de clauses. Par contre, dans le pire cas, contrairement aux algorithmes Removal et Insertion, le nombre d'étapes de HS-C peut être exponentiel dans la taille du nombre de clauses. Donc, il n'est pas surprenant que cet algorithme ait échoué à résoudre plusieurs instances, en particulier quelques grandes instances DIMACS. Cependant, quand il est comparé avec SMUS et CAMUS sur les instances Daimler Chrysler, HS-C a été l'unique algorithme capable de résoudre toutes ces instances, généralement en moins d'une heure. Il est de plus intéressant d'observer les différences

importantes entre les algorithmes HS-C et CAMUS. Comme mentionné dans la Section 3.2.3.3, CAMUS construit d'abord l'ensemble de tous les coMSS et ensuite tous les IIS-C. Quand il réussit, cet algorithme est beaucoup plus rapide que notre méthode HS-C. Cependant, cet algorithme peut échouer (durant la phase 1) si le nombre de coMSS est trop grand ou même (durant la phase 2) si le nombre d'IIS-C est trop grand. Contrairement à CAMUS, notre algorithme HS-C essaie de trouver un IIS de cardinalité minimum en générant seulement un nombre limité de complémentaires de sous-ensembles réalisables (non nécessairement minimaux) : ce sont les ensembles notés $F_S(e_i)$ dans la Figure 4.8. Ceci rend possible la résolution de certaines instances qui sont hors de portée pour CAMUS.

4.6 Conclusion

Nous avons présenté des nouveaux algorithmes permettant de trouver des sous-ensembles incohérents minimaux (IIS) de clauses ou variables d'un problème SAT donné. Nous avons aussi décrit un algorithme qui trouve des IIS avec un nombre minimum de clauses ou variables. Nous avons vu que ces algorithmes d'extraction garantissent la non réalisabilité et la minimalité quand ils sont utilisés avec un algorithme exact, et qu'une garantie de minimalité peut être obtenue, dans certains cas, quand ils sont utilisés avec un algorithme heuristique. De plus, nous avons présenté des techniques additionnelles d'accélération de l'extraction d'un IIS et permettant de trouver des IIS contenant moins de variables et clauses. Finalement, nous avons évalué nos algorithmes en faisant plusieurs expérimentations avec différentes instances, et nous avons comparé nos résultats avec ceux obtenus par d'autres algorithmes existants pour la recherche d'IIS-C : zCore+, AMUSE+, MUP, SMUS, CAMUS. Ces expérimentations ont montré que nos algorithmes ne sont pas les plus rapides, mais ils produisent généralement des IIS avec moins de variables et clauses que les sous-ensembles incohérents obtenus par les algorithmes auxquels nous nous sommes comparés. Pour de très grandes instances sur lesquelles les

algorithmes exacts (p.ex. `zChaff`) ne peuvent pas être appliqués pour prouver leur non réalisabilité, notre algorithme `Insertion` a été le seul capable d'extraire des IIS. Aussi notre algorithme `HS-C` peut déterminer des IIS-C de cardinalité minimum sur des instances qui sont hors de portée pour des algorithmes comme `CAMUS`. En résumé, les algorithmes proposés représentent une alternative intéressante aux algorithmes existants pour l'extraction d'IIS.

CHAPITRE 5

REVUE DE LA LITTÉRATURE CONCERNANT LE FILTRAGE DE CONTRAINTES GLOBALES DE CSP

Dans ce chapitre, nous présentons quelques exemples de méthodes permettant de filtrer les domaines des variables des contraintes globales afin d'obtenir la cohérence de domaine. Tout d'abord, nous rappelons les définitions de contrainte globale et de cohérence de domaine. Les contraintes globales d'un CSP sont les contraintes complexes qui agissent sur plusieurs ou toutes les variables du CSP. Une contrainte globale C agissant sur les variables $x_1, \cdots, x_n$ est domaine-cohérente si pour chaque $i \in [1, \cdots, n]$ et pour chaque valeur $a \in D_i$ il existe $l_1 \in D_1, \ldots, l_{i-1} \in D_{i-1}, l_{i+1} \in D_{i+1}, \ldots, l_n \in D_n$ tels que la contrainte C soit satisfaite.

Comme les contraintes globales sont plus complexes que les contraintes binaires, les algorithmes développés pour obtenir la cohérence de domaine sont généralement créés pour une contrainte globale bien définie, car ils utilisent les particularités et les informations globales de la contrainte pour effectuer le filtrage. Donc, ce ne sont généralement pas des algorithmes génériques pouvant être utilisés pour plusieurs types de contraintes globales.

Dans ce qui suit, nous présentons d'abord la méthode de filtrage de Régin [119] pour la contrainte AllDifferent. Cette méthode a une grande importance en propagation de contraintes, car c'est une des premières méthodes à obtenir la cohérence de domaine pour une contrainte globale d'une façon relativement simple à mettre en place. Ensuite, nous allons présenter le filtrage de Richter et al. [122] pour la contrainte SomeDifferent. Puis, nous allons présenter brièvement d'autres travaux concernant le filtrage des contraintes globales.

5.1 Algorithme de filtrage pour la contrainte AllDifferent

Dans cette section, nous présentons l'algorithme de Régin [119] qui permet de filtrer les domaines des variables afin d'obtenir la cohérence de domaine pour la contrainte AllDifferent.
Les contraintes AllDifferent sont aussi appelées *contraintes de différence*. Rappelons que la contrainte AllDifferent est définie de la façon suivante.

$$\texttt{AllDifferent}(x_1, \ldots, x_n) = \{(a_1, \ldots, a_n) \mid a_i \in D_i \text{ et } a_i \neq a_j \ \forall\, i \neq j\}.$$

Toutes les variables doivent avoir une valeur différente.

Van Hoeve [136] précise qu'il existe au moins six algorithmes de filtrage pour la contrainte AllDifferent, chacun atteignant un niveau différent de cohérence locale ou l'atteignant de façon plus rapide. L'information globale de la contrainte AllDifferent est utilisée par l'algorithme de filtrage de Régin qui traite simultanément toutes les contraintes de non-égalité et qui permet d'obtenir la cohérence de domaine. Cet algorithme est basé sur la théorie du couplage.
Soit X_C l'ensemble de variables sur lesquelles la contrainte de différence est définie. Pour une contrainte de différence C nous pouvons définir un graphe biparti $GV(C) = (X_C, D(X_C), E)$ où $(x_i, a) \in E$ si et seulement si $a \in D_{x_i}$ (i.e., a appartient au domaine de x_i). Ce graphe $GV(C)$ est appelé graphe des valeurs de C (en anglais *value graph*). La Figure 5.1 montre le graphe biparti obtenu si les variables sont $\mathcal{X} = \{x_1, x_2, x_3\}$ et les domaines sont $D_1 = \{1, 3\}$, $D_2 = \{1, 4, 5\}$, $D_3 = \{2, 4\}$.

Définition 5.1.1. Un sous-ensemble d'arêtes d'un graphe $G = (X, E)$ est un *couplage* M si aucune paire d'arêtes n'a un sommet en commun. Un *couplage maximum* est un couplage de cardinalité maximale. Un couplage M dans un graphe G est dit *couvrant* si chaque sommet de X est l'extrémité d'une arête de M.

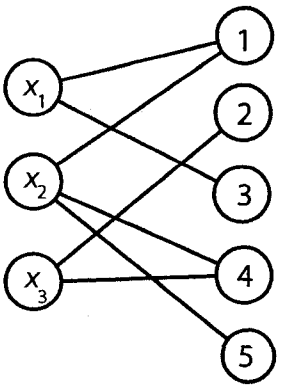


Figure 5.1 – Une contrainte de différence représentée comme un couplage dans un graphe biparti.

Régin [119] donne le théorème suivant concernant la cohérence de la contrainte All-Different.

Théorème 5.1.2. *Soit C une contrainte AllDifferent. Soit $GV(C) = (X_C, D(X_C), E)$ le graphe des valeurs de C. Alors C est domaine-cohérente si et seulement si chaque arête de $GV(C)$ appartient à un couplage couvrant M.*

Régin utilise une propriété énoncée par Berge [19] qui dit qu'une arête appartient à au moins un couplage couvrant si et seulement si, pour un couplage couvrant arbitraire M, cette arête appartient soit à une chaîne alternée paire qui commence à un sommet libre, soit à un cycle alterné pair.

Grâce à cette propriété, Régin énonce la propriété suivante pour la cohérence de domaine.

Proposition 5.1.3. *Soit C une contrainte AllDifferent. Soit une variable x impliquée dans C et soit $d \in D_x$. Soit $GV(C)$ le graphe des valeurs de C et soit M un couplage couvrant de $GV(C)$. Alors le couple (x, d) est supporté par C si et seulement si l'une des conditions suivantes est vérifiée :*

- *l'arête (x, d) appartient à M,*

- *l'arête* (x, d) *appartient à une chaîne alternée paire de* $GV(C)$ *qui commence à un sommet libre,*
- *l'arête* (x, d) *appartient à un cycle alterné pair de* $GV(C)$.

L'exemple de la Figure 5.2 représente le graphe des valeurs de la contrainte All-Different(x_1, x_2, x_3) où $D_1 = \{1, 2\}$, $D_2 = \{1, 2\}$, $D_3 = \{2, 3\}$. Les arêtes en gras représentent un couplage couvrant et l'arête en pointillé représente un couple (x, d) non supporté. En effet, le couple $(x_3, 2)$ n'est pas supporté car il ne peut jamais appartenir à un couplage couvrant. Il faut donc supprimer la valeur 2 du domaine de x_3 pour avoir la cohérence de domaine pour cette contrainte AllDifferent.

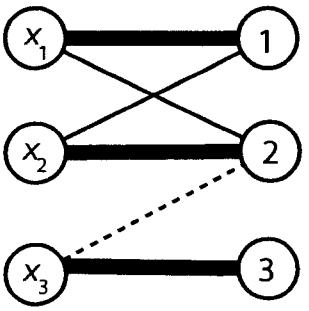


Figure 5.2 – Une contrainte de différence représentée comme un couplage dans un graphe biparti avec en gras les arêtes d'un couplage couvrant et en pointillé une arête représentant un couple (x, d) impossible.

L'algorithme que Régin propose construit le graphe biparti $GV(C)$ décrit ci-dessus et cherche un couplage couvrant les sommets X_C. Ensuite, les arêtes de $GV(C)$ sont orientées de la façon suivante : les arêtes de M sont orientées dans le sens de $D(X_C)$ vers X_C (valeurs vers variables) et les arêtes n'appartenant pas à M sont orientés dans le sens opposé. À partir de cela, il est possible d'identifier les arêtes appartenant à un cycle alterné pair ou à une chaîne alternée paire. Ceci se fait en effectuant une recherche en profondeur d'abord à partir des noeuds libres des $GV(C)$ afin de déterminer les chaînes alternées paires et en cherchant les composantes fortement connexes du graphe ainsi orienté.

Soit $n = |X_C|$ le nombre de sommets de X_C et $m = \sum_{i=1}^{n} |D_i|$, la recherche d'un couplage couvrant peut être fait en $O(m\sqrt{n})$ (montré par Hopcroft et Karp [77]). Donc la vérification de la satisfaisabilité d'une contrainte AllDifferent est en $O(m\sqrt{n})$. La recherche des composantes fortement connexes et des chaînes alternées peut être faite en $O(m+n)$ (montré par Tarjan [132]). Ainsi le filtrage en lui-même (après avoir vérifié la satisfaisabilité de la contrainte) peut être fait en $O(m+n)$. L'avantage de cet algorithme de filtrage est donc que sa complexité est linéaire.
Régin précise que cette méthode est incrémentale. En effet, si certaines valeurs des domaines sont supprimées par le filtrage d'autres contraintes, alors un nouveau couplage peut être calculé en utilisant le précédent.

5.2 Algorithme de filtrage de domaines pour la contrainte SomeDifferent

Nous avons vu qu'une contrainte AllDifferent impose à chaque paire de variables d'avoir une valeur différente. Or, parfois ce ne sont pas toutes les variables qui doivent obtenir des valeurs différentes mais seulement certaines. Richter et al. [122] ont appelé cette contrainte SomeDifferent.
Nous présentons dans cette section l'algorithme de filtrage proposé par Richter et al. [122] afin d'obtenir la cohérence de domaine pour la contrainte SomeDifferent.

Tout d'abord nous rappelons la définition de la contrainte SomeDifferent : $\texttt{SomeDifferent}(x_1, \ldots, x_n) = \{(a_1, \ldots, a_m) \mid a_i \in D_i \; \forall i \text{ et } a_i \neq a_j \; \forall (i,j) \in E\}$.

Richter et al. ont développé un algorithme de filtrage pour ce problème qui énumère les ensembles $U \subset V$ tels que $|D(U)| \leq |U|$ où $D(U) = \bigcup_{v \in U} D_v$. Pour chaque ensemble U et chaque paire (v,i) telle que $v \in V \setminus U$ et $i \in D_v \cap D(U)$, ils vérifient si le couple sommet-couleur (v,i) peut être filtré de G en déterminant si le sous-graphe induit par U est D-colorable quand la valeur i est enlevée de l'ensemble des couleurs des

voisins de v. Ceci peut être fait en transformant le problème de D-coloration de graphe en problème de coloration de graphe classique et en utilisant l'algorithme `Dsatur` de Peemoeller [115]. Comme nous avons aussi appliqué cette transformation dans notre algorithme, la transformation va être présentée en détail dans le Chapitre 6. Pour accélérer leur algorithme, Richter et al. présentent des procédures de simplification. Premièrement, ils simplifient le graphe original en supprimant les arêtes superflues et ensuite, le décomposent en composantes connexes. Ces étapes sont présentées en détail à la section 6.1.2. Dès qu'un couple sommet-couleur (v, i) est détecté comme non supporté, la couleur i est supprimée de D_v.

5.3 Autres travaux concernant le filtrage de contraintes globales

Régin [120] propose un algorithme de filtrage pour la contrainte globale de cardinalité abrégée *gcc* (du terme anglais *global cardinality constraint*). Une contrainte *gcc* est définie sur un ensemble de variables $\mathcal{X} = \{x_1, \ldots, x_n\}$ qui prennent leurs valeurs dans un sous-ensemble de $U = \{u_1, \ldots, u_n\}$. Elle contraint le nombre de fois qu'une valeur $u_i \in U$ est affectée à une variable de $\mathcal{X}$ à être dans un intervalle $[l_i, c_i]$. Cette contrainte s'utilise dans des problèmes d'horaires ou d'ordonnancement (par exemple de voitures sur une chaîne de montage). Afin de pouvoir définir plus formellement cette contrainte nous allons introduire quelques nouvelles notations. Soit $X_{i,\ldots,k}$ un sous-ensemble de $\mathcal{X}$. Un tuple de valeurs des domaines $D_i \times \ldots \times D_k$ est appelé un tuple de $X_{i,\ldots,k}$. Soit $X_C = \{x_i, \ldots, x_k\}$ les variables impliquées dans la contrainte C et $D(C)$ l'union des domaines des variables de X_C, i.e., $D(C) = \bigcup_{j \in X_C} D_j$. Alors, $T(C)$ est l'ensemble des tuples des domaines $D_i \times \ldots \times D_k$ qui sont permis par la contrainte C. Notons $\#(a, t)$ le nombre d'occurrences de la valeur a dans le tuple t.

Définition 5.3.1. *Une contrainte globale de cardinalité est une contrainte C pour laquelle chaque valeur $a_i \in D(C)$ est associée à deux entiers positifs l_i et c_i et telle que*

$T(C) = \{t$ *où* t *est un tuple de* X_C *et* $\forall a_i \in D(C) : l_i \leq \#(a_i, t) \leq c_i\}$.

L'algorithme de filtrage que propose Régin [120] pour cette contrainte est basé sur la théorie des flots. Afin de pouvoir appliquer un algorithme de flots, il construit le réseau orienté suivant.

Définition 5.3.2. *Soit C une contrainte gcc. Le réseau de valeurs de C est un graphe orienté $G(C) = (V, A)$ avec des capacités et des bornes inférieures sur chaque arc. L'ensemble des noeuds de $G(C)$ est formé par l'ensemble X_C, l'ensemble $D(C)$ et deux autres noeuds notés s et t. Les arcs de $G(C)$ sont construits de la façon suivante :*

- *il y a un arc entre une valeur $a \in D(C)$ et une variable $x \in X_C$ si et seulement si $a \in D_x$. Pour ces arcs (a, x) la borne inférieure vaut $l_{ax} = 0$ et la capacité est $c_{ax} = 1$,*
- *il y a un arc entre s et chaque valeur $a_i \in D(C)$. Pour ces arcs (s, a_i), la borne inférieure vaut $l_{sa_i} = l_i$ et la capacité est $c_{sa_i} = c_i$,*
- *il y a un arc entre chaque valeur $x \in X_C$ et t. Pour ces arcs (x, t), la borne inférieure vaut $l_{xt} = 1$ et la capacité est $c_{xt} = 1$,*
- *il y a un arc (t, s) avec $l_{ts} = c_{ts} = |X_C|$.*

Afin de pouvoir déterminer la cohérence d'une contrainte *gcc*, Régin donne la proposition suivante.

Proposition 5.3.3. *Soit C une contrainte gcc telle que $|X_C| = k$ et soit $G(C)$ le réseau de valeurs de C. Alors les propriétés suivantes sont équivalentes :*

- *C est cohérente,*
- *il y a un flot maximum de s vers t dans $G(C)$ de valeur k.*

Étant donné un flot f d'un graphe G, afin de pouvoir déterminer le flot maximum il faut construire le graphe résiduel $R(f)$. Ce graphe est construit de la façon suivante : soit $(u, v) \in A$,

- si $f(u,v) < c_{uv}$ alors $(u,v) \in A(R(f))$ et sa capacité est $r_{uv} = c_{uv} - f(u,v)$,
- si $f(u,v) > l_{uv}$ alors $(v,u) \in A(R(f))$ et sa capacité est $r_{vu} = f(u,v) - l_{uv}$,
- toutes les bornes inférieures sur la capacité sont nulles.

De plus, afin de pouvoir déterminer quelles sont les valeurs des domaines qu'il faut filtrer, il donne la proposition suivante.

Proposition 5.3.4. *Soit C une contrainte gcc cohérente et f un flot maximum dans $G(C)$ de s vers t. Une valeur a d'une variable x n'est pas cohérente avec C si et seulement si $f(a,x) = 0$ et a et x n'appartiennent pas à la même composante fortement connexe de $R(f) \setminus \{(s,t)\}$.*

C'est en se basant sur cette dernière proposition que Régin [120] peut donner un algorithme permettant de filtrer toutes les valeurs incohérentes avec une contrainte *gcc*.

Dans un autre article, Régin [118] présente un algorithme de filtrage pour les contraintes globales de cardinalité avec coûts qui utilise la recherche de flots à coût minimum.

Pesant [116] présente un algorithme de filtrage pour la contrainte *stretch* qui est fréquemment rencontrée dans des problèmes de confection d'horaires de personnel. La contrainte stretch est définie sur une séquence de variables. Elle spécifie des bornes inférieures et supérieures sur le nombre de valeurs consécutives identiques dans cette séquence de variables. L'algorithme de filtrage procède en déterminant des intervalles dans lesquels un stretch donné doit avoir lieu et ensuite, en raisonnant sur ces intervalles, il filtre les valeurs impossibles.

5.4 Conclusion

Dans ce chapitre, nous avons présenté une revue de la littérature de quelques exemples de méthodes de filtrage de contraintes globales de CSP.

Dans le chapitre qui suit (chapitre 6), nous présentons un nouvel algorithme de filtrage pour la contrainte SomeDifferent qui combine un algorithme de recherche tabou avec une méthode exacte.

CHAPITRE 6

ALGORITHME DE FILTRAGE POUR LA CONTRAINTE SOMEDIFFERENT

Rappelons tout d'abord brièvement la définition de la contrainte SomeDifferent. Soit les variables $x_1, \ldots, x_n \in \mathcal{X}$ ayant les domaines respectifs $D_1, \ldots, D_n$ et un graphe $G = (\mathcal{X}, E)$. Alors la contrainte `SomeDifferent` est définie de la façon suivante : $\texttt{SomeDifferent}(x_1, \ldots, x_n) = \{(a_1, \ldots, a_n) \in D_1 \times ... \times D_n \mid a_i \neq a_j \; \forall \, (i,j) \in E\}$. Comme nous l'avons vu dans le chapitre 2, cette contrainte SomeDifferent peut être décrite au moyen d'un problème de D-coloration d'un graphe $G = (V, E)$ où $V = \{1, \ldots, n\}$. Une D-coloration est une fonction $c : V \rightarrow D(V)$ qui une affecte une couleur $c(v) \in D_v$ à chaque sommet $v \in V$ de telle sorte que $c(u) \neq c(v) \; \forall \, (u,v) \in E$. Dans ce chapitre, nous présentons un algorithme permettant de filtrer les domaines des variables pour la contrainte SomeDifferent. Cet algorithme combine une recherche tabou afin de trouver rapidement un support pour le plus de couples sommet-couleur possible, et un algorithme exact afin de valider ou filtrer les couples sommet-couleur restants. Les résultats de ce chapitre ont été présentés dans [50] et [49]. Nous décrivons d'abord de façon générale l'algorithme de filtrage, puis nous présentons en détail les procédures que cet algorithme utilise, ainsi que des techniques d'accélération basées sur des procédures de réduction. À la fin de ce chapitre, nous présentons les résultats expérimentaux que nous avons obtenus.

6.1 Description de l'algorithme de filtrage

Dans cette section, nous décrivons un algorithme de filtrage ayant pour but d'obtenir la cohérence de domaine pour la contrainte SomeDifferent en utilisant sa modélisation comme sous-structure de coloration de graphe. Cet algorithme utilise un algorithme de

recherche tabou appelé `TabuSD` décrit dans la Section 6.1.1. `TabuSD` est utilisé dans deux buts : il peut aider à obtenir une preuve qu'un graphe G donné est D-colorable ; il peut aussi être utilisé pour détecter autant de couples sommet-couleur supportés que possible dans G.

L'algorithme de filtrage utilise aussi des techniques de simplification qui enlèvent des arêtes de G et réduisent les ensembles de couleurs D. Le graphe G' restant associé avec les ensembles de couleurs réduits $D'_v \subseteq D_v$ a la propriété que G est D-colorable si et seulement si G' est D'-colorable. Par conséquent, toutes les couleurs i qui appartiennent à D_v mais pas à D'_v sont telles que (v, i) sont des couples sommet-couleur non supportés et peuvent donc être *filtrés* de G. Ces techniques de simplifications sont rassemblées dans une procédure appelée `Réduction` qui est décrite à la Section 6.1.2.

Un autre algorithme, appelé `TestColorability` a aussi été développé. Celui-ci détermine si un graphe G est D-colorable. Il utilise d'abord l'algorithme `Réduction` mentionné ci-dessus afin de réduire G et D en G' et D'. Comme G est D-colorable si et seulement si chaque composante connexe $G_1, \ldots, G_r$ de G' est D'-colorable, il reste à déterminer si tous les G_j $(j = 1, \ldots, r)$ sont D'-colorables. Afin d'établir ceci, un premier appel à `TabuSD` est effectué afin d'essayer d'obtenir rapidement une preuve de la D'-colorabilité ; si une telle preuve n'est pas obtenue, alors l'algorithme `Dsatur` de Peemoeller [115] est utilisé comme proposé par Richter et al. [122]. La procédure `TestColorability` est décrite en détail à la Section 6.1.3. La valeur retournée par cet algorithme est soit la réponse "NON" si G n'est pas D-colorable, soit la réponse "OUI" associée avec le graphe réduit G' et l'ensemble réduit de couleurs D' dans le cas contraire.

Le schéma général de la procédure de filtrage est donné dans la Figure 6.1. Premièrement, la procédure `TestColorability` est utilisée afin de tester si G est D-colorable.

Si ce n'est pas le cas, alors l'algorithme s'arrête car il n'y a rien à filtrer. Sinon, la valeur retournée par `TestColorability` est un graphe réduit G' ayant les composantes connexes $G_1, \ldots, G_r$ et ayant les ensembles de couleurs réduits D'. `TabuSD` est utilisé pour générer une liste L_j aussi grande que possible de couples sommet-couleur supportés dans chaque composante connexe G_j ($j = 1, \ldots, r$). Ensuite, les couples sommet-couleur (v, i) qui n'appartiennent pas à L_j sont testés afin de déterminer s'ils peuvent être filtrés de G. Ceci est fait en imposant la couleur i au sommet v (i.e., en réduisant D'_v à $D''_v = \{i\}$ avec $D''_u = D'_u \ \forall u \neq v$) et en utilisant `TestColorability` sur G_j avec D''. Si G_j est D''-colorable, alors (v, i) est un couple sommet-couleur supporté dans G_j et peut être ajouté à L_j. Sinon, (v, i) peut être filtré et la procédure `Réduction` est utilisée ce qui peut engendrer le filtrage d'autres couples sommet-couleur.

Algorithme de filtrage

Entrée : Un graphe $G = (V, E)$ avec ensemble de couleurs D.
Sortie : La réponse *NON RÉALISABLE* si G n'est pas D-colorable ; la réponse *RÉALISABLE* et les ensembles de couleurs réduits D' ayant la cohérence de domaine si G est D-colorable.

Appliquer `TestColorability` pour déterminer si G est D-colorable;
si *la réponse est NON* **alors retourner** *NON RÉALISABLE*;
sinon
  Soit $G_1, \cdots, G_r$ les composantes connexes du graphe G' obtenues comme sortie de `TestColorability` avec les ensembles de couleurs réduits D';
  pour chaque $j = 1, \cdots, r$ **faire**
    Appliquer `TabuSD` sur G_j avec les ensembles de couleurs D' pour générer le plus grand ensemble possible L_j de couples sommet-couleur supportés dans G_j ;
    pour chaque *couple sommet-couleur* (v, i) *de* G_j *qui n'appartient pas à* L_j **faire**
      Appliquer `TestColorability` pour déterminer si G_j est D''-colorable avec $D''_v = \{i\}$ et $D''_u = D'_u$ pour $u \neq v$;
      si *la réponse est NON* **alors**
        Enlever i de D'_v;
        Appliquer `Réduction` sur G_j avec ensemble de couleurs D' afin de possiblement filtrer d'autres couples sommet-couleur;
  retourner *(RÉALISABLE,* $D' = (D'_1, \cdots, D'_n)$*)*;

Figure 6.1 – L'algorithme de filtrage.

6.1.1 Une heuristique de recherche tabou

Nous avons développé un algorithme de recherche tabou, appelé `TabuSD` qui peut être utilisé pour essayer de déterminer si un graphe G est D-colorable, ou pour générer le plus grand ensemble possible de couples sommet-couleur dans G. Il est décrit dans la Figure 6.2 et peut être vu comme une extension de l'algorithme `Tabucol` de Hertz et De Werra [75] qui résout le problème classique de k-coloration des sommets d'un graphe (où tous les ensembles de couleurs sont égaux à $\{1, \ldots k\}$ pour un k fixé).
L'espace des solutions $\mathcal{S}$ est l'ensemble de toutes les fonctions $c : V \rightarrow D(V)$ avec $c(v) \in D_v$ pour tout $v \in V$. Par conséquent, une solution n'est pas nécessairement une D-coloration, car des sommets u et v adjacents dans G peuvent avoir la même couleur. Dans une telle situation, on dit que l'arête reliant u à v est une *arête en conflit*. Quand `TabuSD` visite une D-coloration c (i.e., une solution sans arêtes en conflit), toutes les paires $(v, c(v))$ sont introduites dans une liste L qui contient tous les couples sommet-couleur pour lesquels nous avons une preuve qu'ils sont supportés.

Soit $f_1(c)$ le nombre d'arêtes en conflits dans une solution c et $f_2(c)$ le nombre de couples sommet-couleur $(v, c(v))$ qui appartiennent à L. La fonction objectif qui est minimisée par `TabuSD` est définie par

$$f(c) = \alpha f_1(c) + f_2(c)$$

où α est un paramètre qui donne plus ou moins d'importance à la première composante de f. Il est initialement posé égal à 1 et est ensuite ajusté toutes les 10 itérations, comme proposé par Gendreau et al. [56] : si pendant les dix itérations précédentes les solutions étaient toutes des D-colorations de G, alors α est divisé par 2 ; si pendant toutes ces dix itérations il y avait des arêtes en conflit, alors α est multiplié par 2 ; sinon, α reste inchangé.

Une solution voisine $c' \in N(c)$ est obtenue en affectant une nouvelle couleur $c'(v) \neq$

$c(v)$ à exactement un sommet v tel que soit v est adjacent à un sommet u ayant une couleur $c(u) = c(v)$, soit $(v, c(v)) \in L$. Quand un mouvement est effectué pour passer de la solution c à la solution c' en affectant une nouvelle couleur à v, la paire $(v, c(v))$ est déclarée *tabou* ce qui signifie qu'il est interdit de réaffecter la couleur $c(v)$ à v pendant un certain nombre d'itérations. La durée du statut tabou de $(v, c(v))$ est fixée à $K + \lambda\sqrt{|N(c)|}$: si $(v, c(v)) \in L$, $(v, c'(v)) \notin L$, et v n'est adjacent à aucun sommet de couleur $c(v)$, alors K est un entier choisi au hasard dans l'intervalle $[30, 40]$ et on pose $\lambda = 50$; sinon, K est choisi au hasard entre $[20, 30]$ et on pose $\lambda = 1$. Ces valeurs sont différentes selon le cas que l'algorithme rencontre afin que, lorsque $(v, c'(v))$ n'appartient pas à L et $(v, c(v))$ appartenait à L, le statut tabou de l'ancienne solution $(v, c(v))$ soit plus long dans le but de marquer de nouveaux couples sommet-couleur. Ces valeurs semblent raisonnablement bonnes, au moins pour les expériences effectuées et reportées à la section 6.2. Cependant, ces valeurs peuvent être inappropriées pour d'autres instances et un meilleur réglage peut être nécessaire.

Une stratégie de *première amélioration* (first improvement) est utilisée. Plus précisément, quand les solutions dans $N(c)$ sont évaluées, il peut se produire qu'un voisin non tabou c' tel que $f(c') < f(c)$ soit atteint. Dans un tel cas, l'évaluation des autres voisins de c est arrêtée, et le mouvement de c vers c' est effectué. Sinon, `TabuSD` effectue le mouvement de c vers le meilleur voisin non tabou.

Les critères d'arrêt suivants ont été considérés. Si `TabuSD` est utilisé pour détecter le plus de couples sommet-couleur que possible, alors l'algorithme stoppe dès que L contient tous les couples sommet-couleur de G ou que L n'est pas modifié pendant `maxiter` itérations. Si `TabuSD` est utilisé pour essayer de prouver qu'un graphe G est D-colorable, alors l'algorithme stoppe quand une D-coloration est obtenue (i.e., une solution c avec $f_1(c) = 0$). Pour les expérimentations, nous avons posé `maxiter`=2000.

TabuSD

Entrée : Un graphe $G = (V, E)$ avec ensemble de couleurs D
Sortie : Un ensemble L de couples sommet-couleur supportés

1 **Initialisation**
2 Générer une solution initiale c en choisissant une couleur $c(v) \in D_v$ pour chaque sommet v;
3 Poser $\alpha \leftarrow 1$ et $L \leftarrow \emptyset$;
4 **tant que** *aucun critère d'arrêt n'est rencontré* **faire**
5 Poser $f_{best} \leftarrow +\infty$;
6 **pour chaque** *sommet v qui est l'extrémité d'une arête en conflit ou tel que* $(v, c(v)) \in L$ **faire**
7 **pour chaque** *couleur* $i \in D_v \backslash \{c(v)\}$ **faire**
8 **si** (v, i) *n'est pas un mouvement tabou* **alors**
9 Soit c' le voisin de c obtenu en affectant la couleur i à v;
10 **si** $f(c') < f_{best}$ **alors** poser $v_{best} \leftarrow v$, $i_{best} \leftarrow i$ et $f_{best} \leftarrow f(c')$;
11 **si** $f(c') < f(c)$ **alors** aller ligne 12;
12 Introduire $(v_{best}, c(v_{best}))$ dans la liste tabou;
13 Modifier c en affectant la couleur i_{best} à v_{best};
14 **si** $f_1(c) = 0$ **alors**
15 Poser $L \leftarrow L \cup \bigcup_{v \in V} (v, c(v))$;
16 Mettre à jour α ;
17 **retourner** L

Figure 6.2 – L'algorithme TabuSD.

6.1.2 Les procédures de réduction

Avant d'appliquer un algorithme pour filtrer tous les couples sommet-couleur non supportés d'un graphe G ayant les ensembles de couleurs D, il peut être utile d'employer des techniques de simplifications qui peuvent possiblement enlever des arêtes de G et réduire les ensembles de couleurs.

Une arête est *superflue* si ses deux extrémités ont des ensembles de couleurs disjoints. Comme mentionné par Richter et al. [122], il est évident que la suppression des arêtes superflues de G ne modifie pas l'ensemble de couples sommet-couleur supportés.

De plus, quand l'ensemble de couleurs d'un sommet v contient une seule couleur i, il est possible d'appliquer une vérification vers l'avant (*forward checking*) sur les contraintes binaires de non égalité. Cela signifie que la couleur i peut être enlevée (i.e., filtrée) des ensembles de couleurs des sommets u adjacents à v, à nouveau ceci ne modifie pas l'ensemble des couples sommet-couleur supportés. Comme toutes les arêtes incidentes à v deviennent superflues, elles peuvent être enlevées de G.

En appliquant les transformations mentionnées ci-dessus aussi souvent que possible, on obtient un graphe G' ayant les ensembles de couleurs D' et telle que G est D-colorable si et seulement si G' est D'-colorable. Soit $G_1, \ldots, G_r$ les composantes connexes de G. Comme observé par Richter et al. [122], il est clair que G' est D'-colorable si et seulement si chaque G'_i est D'-colorable.

La procédure qui transforme G et D en G' et D' est appelée `Réduction`. Elle est utilisée par Richter et al. [122] sans la réduction sur les ensembles de couleurs qui sont des singletons. Il est à remarquer que si un ensemble de couleurs D'_v est vide alors G' n'est pas D'-colorable. Dans un tel cas, G n'a aucun couple sommet-couleur supporté et

la procédure `Réduction` peut alors être arrêtée.

Dans la Figure 6.3 est présentée une illustration de cette procédure `Réduction`. Dans la figure (a), le sommet H a une seule couleur possible : 1. Alors cette couleur est supprimée des domaines des sommets adjacents à H, i.e., F, G, I et J et les arêtes (F, H), (G, H), (H, I) et (H, J) deviennent superflues et sont supprimées. Ceci produit le nouveau graphe présenté en (b). Nous pouvons voir que les sommets F et G ont seulement la couleur 3 possible, donc cette couleur est supprimée des domaines de leurs voisins, i.e., D et E, et les arêtes (D, G) et (E, H) deviennent superflues et sont supprimées. Le domaine du sommet I contient uniquement la couleur 2. Comme cette couleur n'appartient pas au domaine de l'unique voisin de I, i.e., J, nous ne modifions pas le domaine de J par contre l'arête (I, J) devient superflue et peut donc être supprimée. Les domaines des sommets J et K ont une intersection vide, donc nous pouvons supprimer l'arête (J, K). Le graphe réduit est présenté en (c). En résumé, dans cet exemple, les couples sommet-couleur qui sont supprimés sont $(F, 1), (G, 1), (I, 1), (J, 1), (D, 3)$ et $(E, 3)$. Tous les autres couples sommet-couleur (v, i) sont supportés.

6.1.3 Test de colorabilité

Nous avons développé un algorithme, appelé `TestColorability`, qui détermine si un graphe G est D-colorable. Cette procédure décrite dans l'Algorithme 6.6 utilise premièrement la procédure `Réduction` de la Section 6.1.2 pour réduire G et D en G' et D'. Nous déterminons ensuite les composantes connexes $G_1, \ldots, G_r$ de G'.

Remarquons que si un ensemble de couleurs D'_v est vide, alors G' n'est pas D'-colorable, ce qui signifie que G n'est pas D-colorable. Alors, supposons qu'aucun ensemble de couleurs D'_v n'est vide. Comme G est D-colorable si et seulement si chaque composante connexe $G_1, \ldots, G_r$ est D'-colorable, il reste à déterminer si tous les G_j $(j = 1, \ldots, r)$

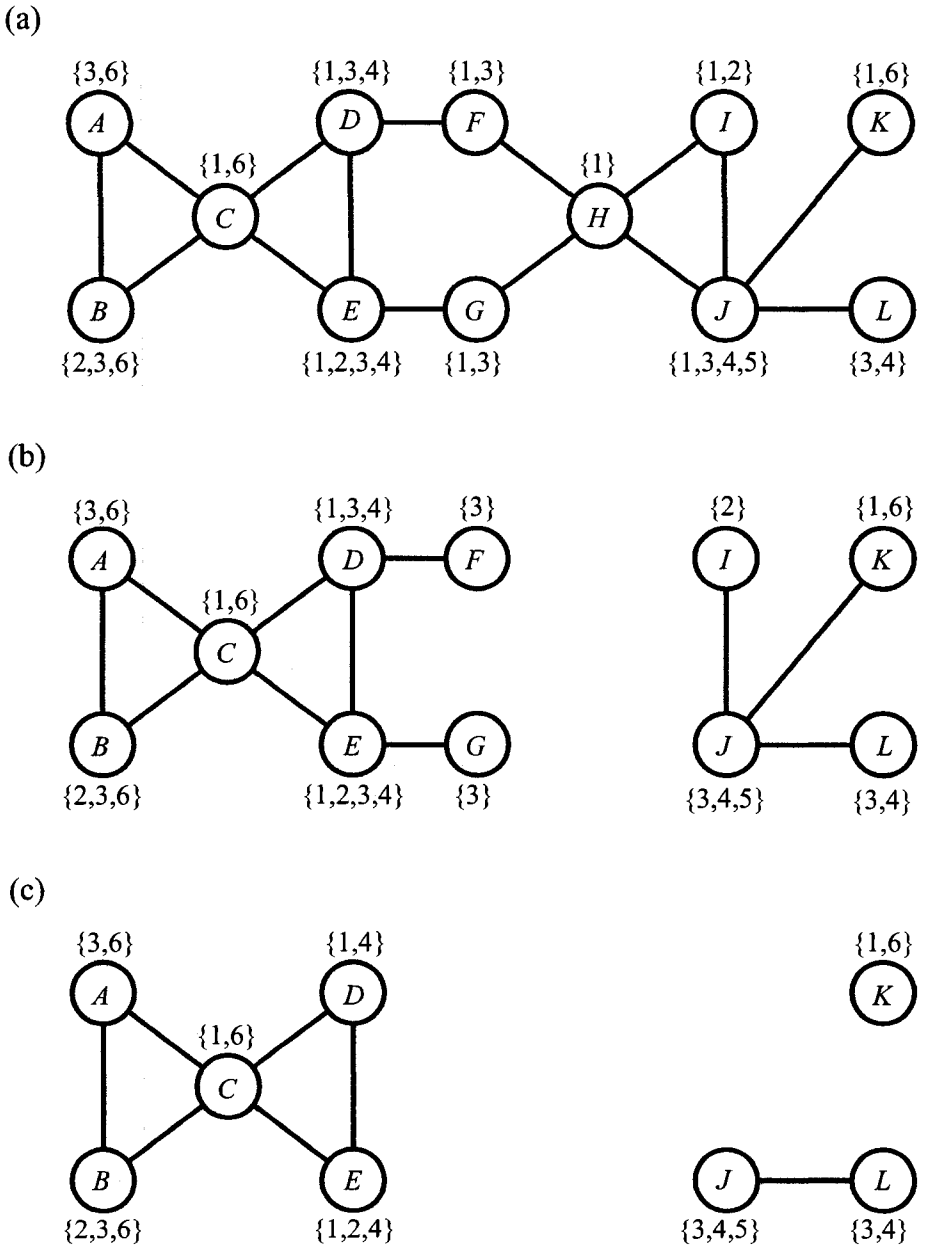


Figure 6.3 – Illustration de l'algorithme `Réduction`.

sont D'-colorables. Si la liste L obtenue comme sortie de `TabuSD` n'est pas vide, alors nous savons que G_j est D'-colorable. Sinon, nous allons résoudre un problème de coloration de graphe classique qui est décrit ci-dessous.

Étant donné un graphe G avec l'ensemble de sommets V et l'ensemble d'arêtes E, le problème classique de coloration de graphe consiste à affecter une couleur à chaque sommet telle que deux sommets adjacents obtiennent des couleurs différentes et le nombre total des différentes couleurs est minimisé. Ce nombre minimum requis de couleurs est appelé le *nombre chromatique* de G et est noté $\chi(G)$. C'est un cas particulier de notre problème de D-coloration car pour chaque entier positif k nous pouvons définir $D_v = \{1, \ldots, k\}$, et nous avons $\chi(G) \leq k$ si et seulement si G est D-colorable. Notre but est donc de transformer le problème de D-coloration en un problème classique de coloration de graphe. Ceci est fait comme Richter et al. [122]. Étant donné un graphe G avec ensemble de sommets V et ensembles de couleurs D, nous construisons un nouveau graphe $G \oplus D$ à partir de G en ajoutant un ensemble de $|D(V)|$ nouveaux sommets tous reliés par une arête (i.e., formant une clique), chacun de ces nouveaux sommets correspondant à une couleur dans $D(V)$, et en reliant chaque sommet $v \in V$ à un nouveau sommet i si et seulement si $i \notin D_v$. Il est alors facile d'observer que G est D-colorable si et seulement si $\chi(G \oplus D) = |D(V)|$. Pour déterminer $\chi(G \oplus D)$, nous utilisons l'algorithme `Dsatur` [115] qui peut être téléchargé à partir du site web de Michael Trick [134].

Dans la Figure 6.4 est présenté un graphe $G = (V, E)$ que nous voulons transformer en $G \oplus D$. Dans ce cas $D(V) = \{1, 2, 3\}$ et donc $|D(V)| = 3$. Donc il faut rajouter une clique composée de 3 sommets. Ces nouveaux sommets vont porter les étiquettes $a1$, $a2$ et $a3$. Le sommet $a1$ correspond à la couleur 1 de $D(V)$, le sommet $a2$ correspond à la couleur 2 et le sommet $a3$ à la couleur 3. Ainsi, le sommet $a2$ va être relié aux sommets 2 et 4 (car ils n'ont pas la couleur 2 dans leur domaine) et le sommet $a3$ va être relié aux sommets 1 et 3 (car ils n'ont pas la couleur 3 dans leur domaine). Le nouveau graphe $G \oplus D$ est présenté dans la Figure 6.5. Le nombre chromatique de $G \oplus D$ vaut 3. En effet,

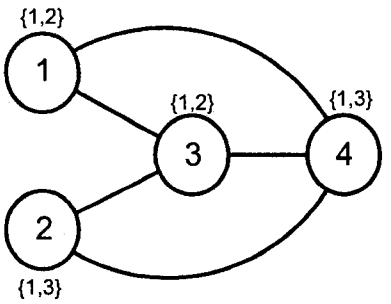


Figure 6.4 – Un graphe G avant la transformation pour Dsatur

la coloration c suivante est légale : $c(1) = 1$, $c(2) = 1$, $c(3) = 2$, $c(4) = 3$, $c(a1) = 1$, $c(a2) = 2$ et $c(a3) = 3$. Comme $\chi(G \oplus D) = |D(V)|$ le graphe G est D-colorable.

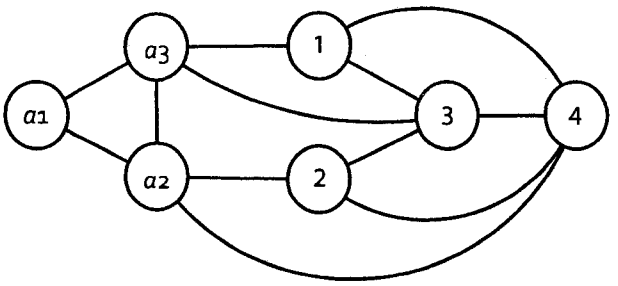


Figure 6.5 – Le graphe $G \oplus D$ où G est le graphe de la Figure 6.4.

Soit G_j une composante connexe du graphe produit comme sortie de la procédure `Reduction`. Si G_j contient un seul sommet v, alors G_j est D'-colorable car v peut avoir n'importe quelle couleur de D'_v. Si G_j contient deux sommets v et u, alors D'_v et D'_u ne sont pas des singletons car la procédure `Réduction` isole chaque sommet avec une seule couleur dans son ensemble de couleurs. Par conséquent, pour n'importe quel choix de couleur $i \in D'_v$ pour v, il y a une couleur différente de i dans D'_u qui peut être affectée à u, ce qui signifie que G_j est D'-colorable. Donc la transformation en un problème de coloration classique décrite ci-dessus n'est utilisée que si G_j contient au

moins trois sommets.

Algorithme 21 : TestColorability

Entrée : Un graphe $G = (V, E)$ avec ensembles de couleurs D.
Sortie : La réponse NON si G n'est pas D-colorable ; la réponse OUI associée avec un graphe réduit G' et des ensembles de couleurs réduits D' sinon.

Appliquer `Réduction` sur G et D, et soit G' et D' le graphe réduit et les ensembles de couleur réduits obtenus comme sortie;
si *un ensemble de couleur* D'_v *est vide* **alors retourner** *NON*;
sinon
  Déterminer les composantes connexes $G_1, \cdots, G_r$ de G';
  pour chaque G_j *avec au moins trois sommets* **faire**
    Appliquer `TabuSD` pour essayer de déterminer si G_j est D'-colorable;
    si *la liste de sortie* L *est vide* **alors**
      Construire $G \oplus D$ et utiliser `Dsatur` pour déterminer $\chi(G \oplus D)$;
      si $\chi(G \oplus D) > |\bigcup_{v \in G_j} D'(v)|$ **alors**
        retourner *NON* et STOP;
  retourner *(OUI, G', D');*

Figure 6.6 – La procédure TestColorability.

6.2 Résultats expérimentaux

L'algorithme a été évalué sur quatre types d'instances : des données réelles, des graphes aléatoires, des graphes ayant une unique D-coloration et des graphes modélisant des problèmes de SUDOKU. Tous les tests ont été effectués sur un Pentium D 2.80GHz avec 1024K de cache fonctionnant avec Linux Centos 2.6.9. Comme l'algorithme de filtrage proposé utilise des heuristiques, cinq relances ont été faites sur chaque instance. Les résultats sont décrits dans les sections ci-dessous.

6.2.1 Données provenant d'un problème réel de planification de la main-d'oeuvre

Le problème réel étudié par Richter et al. [122] est un problème de planification de la main-d'oeuvre pour un département d'IBM. Les données de ce problème sont les suivantes : nous avons un ensemble de tâches ayant des dates durant lesquelles chaque tâche doit être effectuée, et une liste de personnes qualifiées pour effectuer ces tâches. Les tâches qui se chevauchent dans le temps ne peuvent pas être effectuées par la même personne. Ceci est un problème `SomeDifferent` typique qui peut être modélisé avec un problème de D-coloration où les tâches sont les sommets du graphe, les couleurs sont les personnes, et il y a une arête entre deux sommets si les tâches correspondantes se chevauchent. L'ensemble de couleurs D_v d'un sommet v est l'ensemble des personnes qualifiées pour effectuer v. Au total, il y a 290 instances avec un nombre n de tâches variant de 20 à 300.

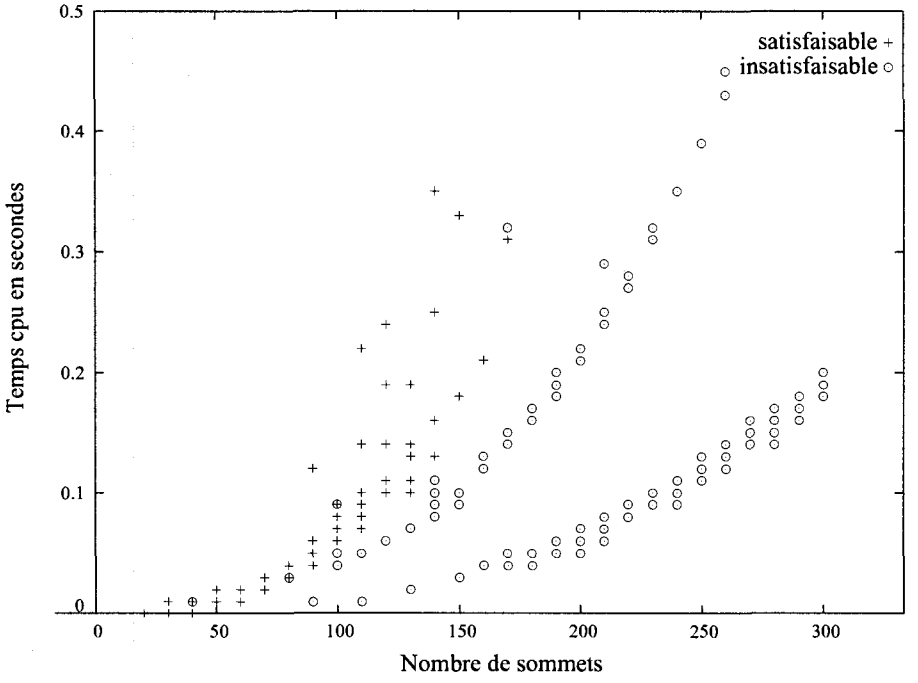


Figure 6.7 – Temps d'exécution pour les données de planification de la main-d'oeuvre.

Les temps d'exécution de l'algorithme de filtrage sont rapportés dans la Figure 6.7.

Les instances réalisables sont représentées avec un signe '+', et les non réalisables avec un signe 'o'. Nous observons que beaucoup d'instances (en particulier toutes celles avec plus de 170 tâches) sont non réalisables. Pour les instances non réalisables nous pouvons identifier deux courbes. La courbe inférieure contient les instances pour lesquelles un ensemble de couleur vide a été généré par la procédure `Réduction` dans `TestColorability`. La courbe supérieure, qui a une croissance exponentielle, contient les instances pour lesquelles un appel à l'algorithme `Dsatur` a été requis pour avoir une preuve de non satisfaisabilité. Bien que la courbe pour les instances réalisables ait aussi une croissance exponentielle, nous observons que tous les temps d'exécutions n'excédent jamais 0.35s. Des courbes similaires avaient été reportées par Richter et al. [122].

Nous avons aussi effectué des tests sur les instances réalisables sans aucun appel à `TabuSD` (i.e., en utilisant uniquement des appels à `Dsatur` pour chaque couple sommet-couleur existant) afin de quantifier l'aide fournie par l'utilisation de la recherche locale dans le processus de filtrage. Les temps cpu ont augmenté jusqu'à 10.61s et étaient en moyenne 34 fois plus lents.

Dans la Figure 6.8 sont présentés deux histogrammes pour les instances réalisables avec $n = 30, 60, 100, 170$. Ces instances ont très peu de couples sommet-couleur non supportés, à savoir 0% pour $n = 30$, 0.02% pour $n = 60$, 0.2% pour $n = 100$, et 0.64% pour $n = 170$. Les couples sommet-couleur (v, i) supportés peuvent être détectés de trois différentes façons, et leur distribution est montrée dans le premier histogramme : si v appartient à une composante connexe comprenant moins de trois sommets, alors toutes les couleurs i dans D_v définissent un couple sommet-couleur (v, i) supporté (et l'étiquette "$\in$ CC < 3" est utilisée) ; quand une couleur $i \in D_v$ n'apparaît dans aucun ensemble de couleurs D_u avec u adjacent à v, alors (v, i) est un couple sommet-couleur supporté (et l'étiquette "$i \notin D_u$" est utilisée) ; la troisième possibilité est d'obtenir une preuve que

(v, i) est supportée avec TabuSD (et l'étiquette "avec TabuSD" est utilisée).

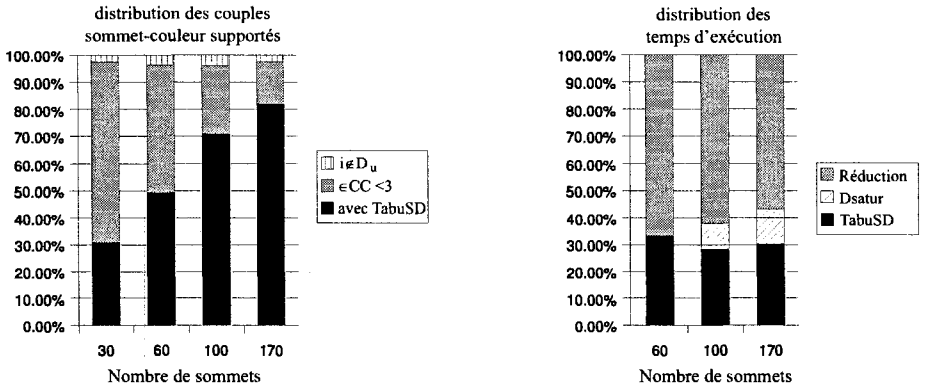


Figure 6.8 – Statistiques sur les couples sommet-couleur supportés et les temps d'exécution pour quelques données provenant du problème de planification de la main-d'oeuvre.

Nous observons que la plupart des couples sommet-couleur (v, i) sont montrés supportés avec TabuSD ou avec la procédure Réduction qui crée des composantes connexes avec moins de trois sommets. La détection grâce à TabuSD augmente avec la taille de l'instance. Le deuxième histogramme indique la distribution des temps d'exécution passés soit dans la procédure Réduction, dans TabuSD ou dans Dsatur. Nous ne rapportons pas les temps d'exécution pour $n = 30$ parce qu'ils sont trop petits (typiquement moins de 0.001s). Nous observons que TabuSD consomme environ 30% du temps total d'exécution. Le temps passé dans la procédure Réduction diminue un petit peu (de 66.7% à 56.8%) pendant qu'il augmente pour Dsatur (de 0% à 12.9%) quand le nombre de sommets augmente de 60 à 170.

6.2.2 Graphes aléatoires

Pour notre deuxième ensemble de tests, nous considérons les mêmes graphes aléatoires que Richter et al. [122]. Ces graphes sont définis par un quadruplet (n, p, d, max_k) de paramètres : n est le nombre de sommets, p est la probabilité (selon une distribution uniforme) d'avoir une arête entre deux sommets, $d = |D(V)|$ est le nombre total de couleurs différentes, et max_k est la taille maximale d'un ensemble de couleurs.

Pour chaque sommet v, un nombre entier k_v est choisi uniformément dans l'intervalle $[1, \ldots, max_k]$, et l'ensemble de couleur D_v pour v est généré en choisissant aléatoirement k_v différentes couleurs dans l'intervalle $[1, \ldots, d]$. Les instances de Richter et al. [122] ont $n = 20, 30, \ldots, 100$, $p = 0.1, 0.3, 0.6$, $d = 300$, et $max_k = 10, 20$. Nous avons généré des instances additionnelles en considérant aussi $n = 200, 500$, $p = 0.9$, et $max_k = 5, 40, 80$.

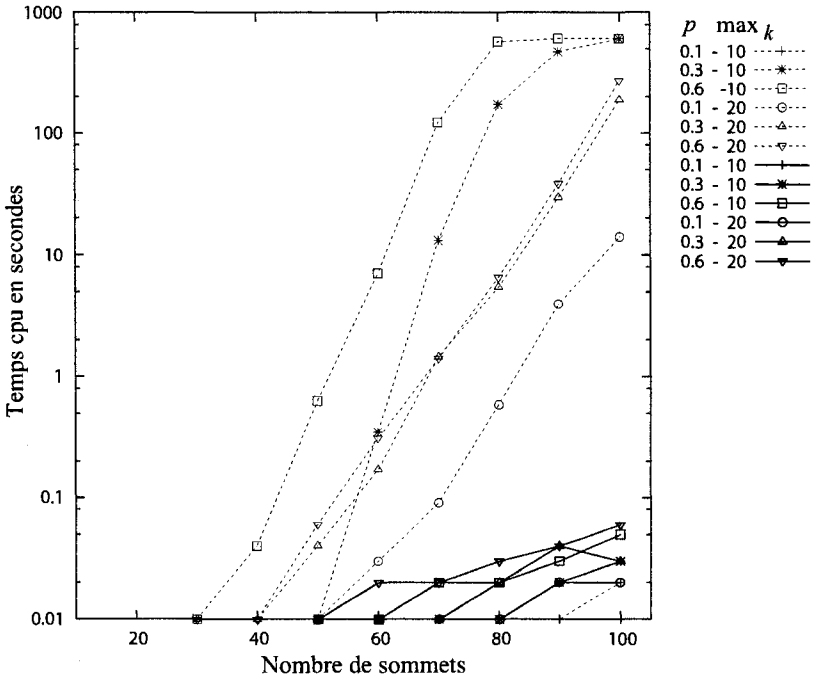


Figure 6.9 – Comparaison de nos temps d'exécution avec ceux de Richter et al. [122] pour les instances aléatoires.

Dans la Figure 6.9, nous comparons nos temps d'exécution (représentés par des lignes pleines) avec ceux rapportés par Richter et al. [122] (représentés par des pointillés) obtenus avec un ordinateur qui est un petit peu plus rapide que le nôtre. L'échelle de temps est logarithmique. Nous observons que notre méthode est beaucoup plus rapide car le temps cpu maximum rapporté par Richter et al. est 608.73s (pour $n = 100$, $p = 0.6$, $max_k = 10$) alors que notre algorithme de filtrage ne requiert jamais plus de 0.058s. Il n'est pas surprenant que les instances les plus difficiles sont celles avec $p = 0.6$ car la

procédure `Réduction` produit dans ce cas-là un graphe réduit avec moins de composantes connexes, ce qui demande l'application de `TabuSD` et `Dsatur` sur des graphes plus grands.

Dans la Figure 6.10, nous présentons les mêmes histogrammes que dans la Figure 6.8, mais pour les graphes aléatoires. La lettre "A" indique les instances avec $max_k = 10$ et $p = 0.1$, "B" les instances avec $max_k = 20$ et $p = 0.3$, et "C" les instances avec $max_k = 20$ et $p = 0.6$. À nouveau, le pourcentage de couples sommet-couleur non supportés est très petit, avec un maximum de 0.6% pour les instances C avec $n = 100$. Nous observons que pour $max_k = 10$ et $p = 0.1$ (i.e., les instances du type A), beaucoup de couples sommet-couleur (v, i) sont montrés supportés parce que v appartient à une composante connexe de taille au plus deux. De plus, pour chaque type de paramètre, A, B, ou C, le nombre de couples sommet-couleur montrés supportés par `TabuSD` augmente avec la taille des instances. Le deuxième histogramme ne rapporte aucune statistique pour $n = 40$ car les temps d'exécution sont trop petits. Notons qu'il n'y a pas d'appel à `Dsatur`. Ceci est dû au fait que tous les couples sommet-couleur supportés sont détectés par la procédure `Réduction` ou par `TabuSD`.

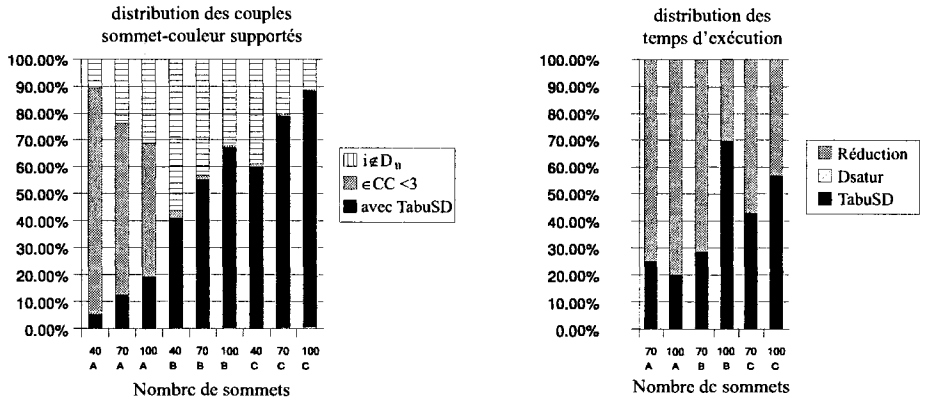


Figure 6.10 – Statistiques sur les couples sommet-couleur supportés et les temps d'exécution pour les graphes aléatoires. A pour les paramètres $max_k = 10, p = 0.1$, B pour $max_k = 20, p = 0.3$, et C pour $max_k = 20, p = 0.6$.

Dans la Figure 6.11, nous rapportons tous nos temps d'exécution pour les graphes aléatoires, incluant ceux pour $n = 200, 500$, $p = 0.9$, et $max_k = 5, 40, 80$. À nouveau, l'échelle de temps est logarithmique. Le temps cpu maximum pour les instances ayant jusqu'à 100 sommets est 0.2s, ce qui peut être considéré comme raisonnable pour un algorithme utilisé dans le but d'obtenir la cohérence de domaine. Pour les graphes aléatoires avec 200 sommets, le temps cpu maximum augmente jusqu'à 1.5s cpu, et il atteint 41.5s pour $n = 500$. L'augmentation du temps cpu est principalement due à l'utilisation de `TabuSD` qui requiert beaucoup d'itérations pour trouver un support pour presque tous les couples sommet-couleur. Pour tous ces graphes aléatoires, nous avons observé que presque tous les couples sommet-couleur qui peuvent être filtrés sont détectés durant le premier appel à la procédure `Réduction`.

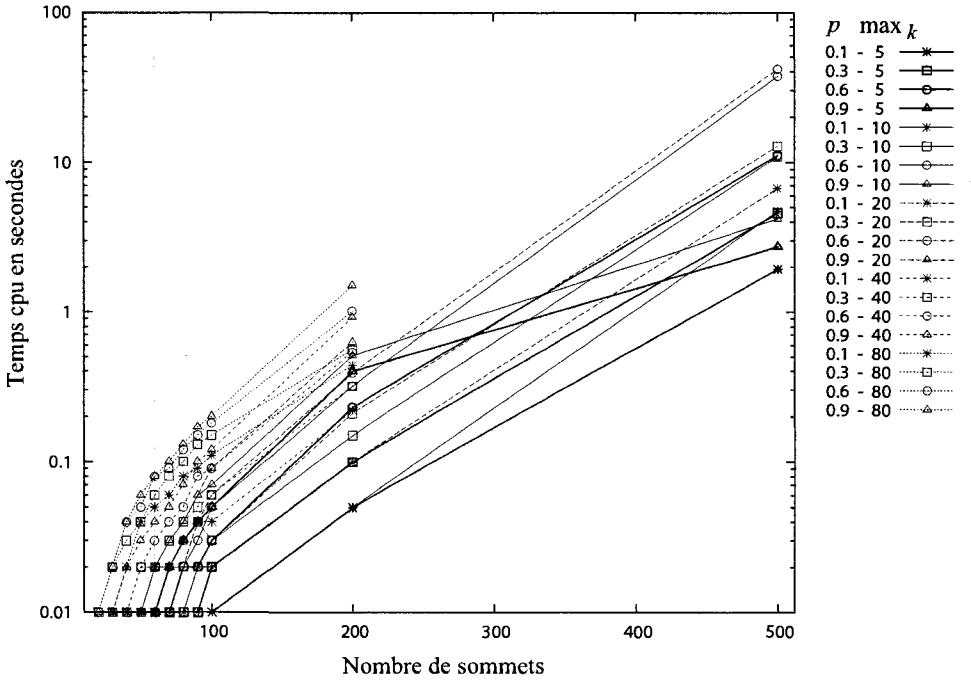


Figure 6.11 – Temps d'exécution pour notre algorithme de filtrage pour tous les graphes aléatoires.

6.2.3 Graphes avec une unique D-coloration

Mahdian et Mahmoodian [99] ont décrit une famille de graphes avec une unique D-coloration. Plus précisément, étant donné un entier k, ils définissent un graphe avec $3k-2$ sommets, $\frac{1}{2}(9k^2-17k+8)$ arêtes, et pour lesquels chaque ensemble de couleurs D_v contient exactement k couleurs. Pour chaque sommet v, il y a exactement un couple sommet-couleur (v,i) qui est supporté avec $i \in D_v$, ce qui signifie que $(k-1)(3k-2) = 3k^2-5k+2$ couples sommet-couleur peuvent être filtrés. Des exemples de tels graphes pour $k=2$ et $k=3$ sont présentés dans les Figures 6.12 et 6.13. Dans chacun des cas, le graphe de gauche représente le graphe original avec pour chaque sommet v l'ensemble de couleurs D_v et le graphe de droite représente le graphe filtré avec les ensembles de couleurs réduits D'_v.

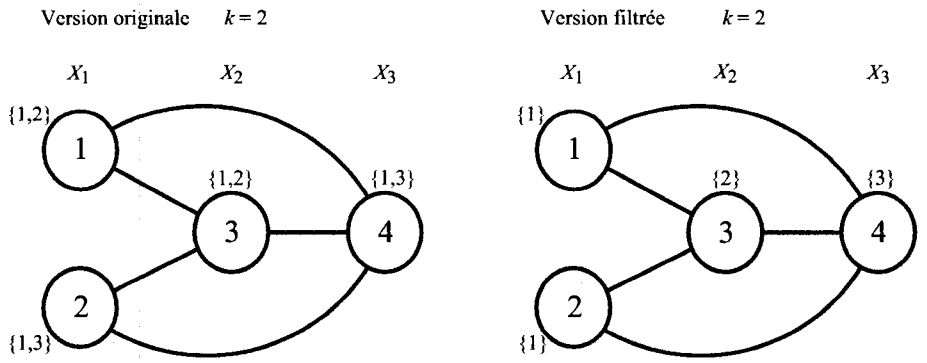


Figure 6.12 – Graphe ayant une unique D-coloration tel que décrit par Mahdian et Mahmoodian [99] pour $k=2$. Le graphe de gauche est la version originale du graphe et le graphe de droite est la version filtrée.

Les temps d'exécution pour $k = 5, 6, 7, 8, 9$ sont rapportés dans la Figure 6.14. Ces instances sont difficiles, principalement parce que la procédure `Réduction` n'est pas capable de réduire le graphe original et les ensembles des couleurs originaux, ce qui signifie que `Dsatur` est requis pour confirmer que les couples sommet-couleur non supportés peuvent bien être filtrés. La croissance exponentielle des temps d'exécution qui excèdent 100s pour des graphes avec 25 sommets ($k=9$) montre la limite de l'applicabilité de notre algorithme de filtrage.

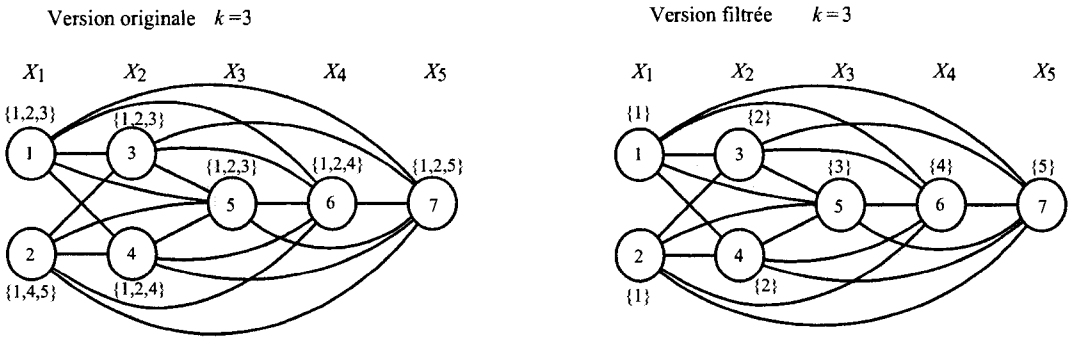


Figure 6.13 – Graphe ayant une unique D-coloration tel que décrit par Mahdian et Mahmoodian [99] pour $k = 3$. Le graphe de gauche est la version originale du graphe et le graphe de droite est la version filtrée.

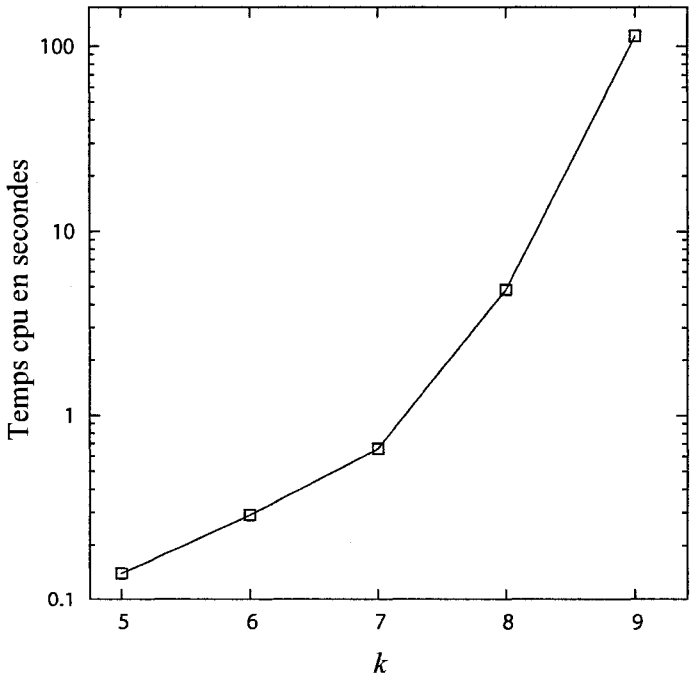


Figure 6.14 – Temps d'exécution sur les graphes ayant une unique D-coloration.

Le premier histogramme de la Figure 6.15 indique le pourcentage de couples sommet-couleur qui sont soit filtrés, soit montrés supportés par `TabuSD` ou `Dsatur`. Comme attendu, la proportion de couples sommet-couleur filtrés est $\frac{k-1}{k}$. Alors que la plupart de couples sommet-couleur sont montrés supportés avec `TabuSD` pour des petites valeurs de k, `Dsatur` fait le travail pour des plus grandes valeurs car `TabuSD` n'est pas capable de trouver l'unique D-coloration (avec `maxiter=2000`). Des statistiques sur les temps d'exécution sont données dans le deuxième histogramme où nous pouvons clairement observer que le temps passé dans `Dsatur` augmente radicalement avec k.

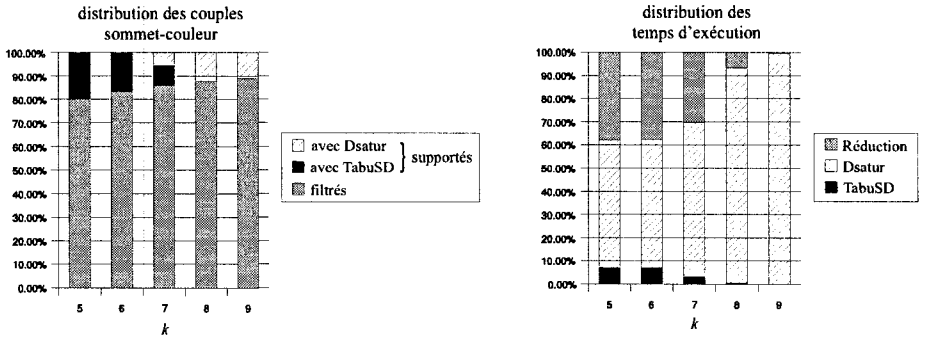


Figure 6.15 – Statistiques sur les couples sommet-couleur supportés et non supportés et sur les temps d'exécution pour des graphes avec une unique D-coloration.

Avec l'espoir de générer des graphes avec un nombre significatif mais plus réaliste de couples sommet-couleur à filtrer, nous avons créé des variations de ces graphes en supprimant un certain pourcentage p d'arêtes. Nous avons effectué des tests pour $k = 5, 6, 7, 8, 9$ et $p = 0, 0.002, 0.005, 0.1, 0.15$ et pour chaque paire de paramètres (exceptés pour $p = 0$ où le graphe est unique), nous avons généré dix graphes. Les résultats obtenus sur ces graphes ne sont pas homogènes, donc nous ne les présenterons pas en détail. Par exemple, alors que 113s étaient nécessaires pour résoudre le graphe original pour $k = 9$ (et $p = 0$), le temps nécessaire pour les graphes avec $k = 9$ et $p = 0.05$ s'étalaient de 0.32 secondes à 182.7 secondes. De plus, pour $p = 0.15$, aucun couple sommet-couleur ne pouvait être filtré, ce qui allait à l'encontre du but recherché.

6.2.4 Graphes modélisant des problèmes de SUDOKU

Une application amusante de notre algorithme de filtrage permet de trouver l'unique solution des problèmes de SUDOKU (quand ils ont bien une seule solution ; dans les cas de non-unicité de la solution, notre algorithme permet de réduire les domaines de valeurs possibles de chaque case).

Rappelons d'abord ce qu'est un problème de type SUDOKU. C'est un problème sur une grille comprenant 9×9 (pour la version originale) cases à remplir avec les chiffres de 1 à 9. La grille est subdivisée en 9 sous-grilles de 3×3 cases (les sous-grilles sont délimitées par des doubles lignes dans les figures qui suivent). Chaque chiffre de 1 à 9 ne peut apparaître qu'une seule fois dans chaque colonne, dans chaque ligne et dans chaque sous-grille. À l'origine la grille est partiellement remplie et le but est de la remplir totalement. Selon le nombre de cases remplies au départ et la position de ces cases, le problème est plus ou moins facile à résoudre.
Ce jeu se modélise facilement avec un graphe. Chaque case est un sommet du graphe et entre chaque sommet modélisant une case d'une même ligne, d'une même colonne ou d'une même sous-grille il y a une arête. Le domaine de chaque sommet est composé des couleurs 1 à 9, sauf pour les cases remplies au départ pour lesquelles le domaine est réduit à la valeur donnée. Le graphe comporte donc 81 sommets et 810 arêtes. Pour chaque sommet, il y a un seul couple sommet-couleur qui est supporté. Nous avons testé trois instances de difficultés différentes : une facile, une moyenne et une très difficile. Les trois ont une seule solution possible.

Le problème facile est présenté dans la figure 6.16.

Cette instance comporte 473 couples sommet-couleurs à tester et il y en seulement 81 paires possibles. Lors de la résolution de cette instance avec notre algorithme de filtrage toutes les valeurs impossibles sont filtrées par le prétraitement effectué par la

	4		**6**				**8**	
6		**7**		**9**				
				2	**7**	**5**		**3**
		8	**3**					**4**
	2	**4**	**8**		**9**	**3**	**5**	
9					**1**	**8**		
2		**9**	**1**	**5**				
				8		**1**		**5**
	1				**6**		**9**	

Figure 6.16 – Problème SUDOKU de type facile

procédure `Réduction` et la seule solution possible est donc immédiatement obtenue. Il n'y a donc aucun appel à l'algorithme de recherche tabou pour marquer la seule solution possible. Le temps total obtenu est 0.03s (moyenne en utilisant cinq graines différentes). La solution est présentée dans la Figure 6.17.

5	**4**	2	**6**	1	3	7	**8**	9
6	3	**7**	5	**9**	8	4	1	2
8	9	1	4	**2**	**7**	**5**	6	**3**
1	6	**8**	**3**	7	5	9	2	**4**
7	**2**	**4**	**8**	6	**9**	**3**	**5**	1
9	5	3	2	4	**1**	**8**	7	6
2	8	**9**	**1**	**5**	4	6	3	7
3	7	6	9	**8**	2	**1**	4	**5**
4	**1**	5	7	3	**6**	2	**9**	8

Figure 6.17 – Résolution du problème SUDOKU de type facile

L'instance de difficulté moyenne que nous avons testée est présentée dans la Figure 6.18.

Cette instance comporte au départ 505 couples sommet-couleur possibles et bien entendu il y en aussi seulement 81 possibles. Il y a 28 cases qui sont remplies à l'origine.

				6	**3**	**7**		**8**
8		**2**						**6**
		7		**9**				
	5		**9**				**3**	
	1		**8**		**5**		**9**	
	7				**6**		**8**	
				8		**1**		
1						**8**		**3**
3		**6**	**5**	**2**				

Figure 6.18 – Problème SUDOKU de type moyen.

Les 28 couples sommet-couleur correspondantes sont marquées par le prétraitement de la procédure `Réduction`. De plus, cette procédure permet aussi de filtrer 302 paires et de supprimer 489 arêtes. Après la décomposition en composantes connexes il y a 29 composantes connexes : 28 correspondent aux sommets modélisant les cases remplies au départ et la dernière comporte tous les autres sommets. Le tabou permet de marquer les 53 autres paires possibles et Dsatur, ainsi que les réductions effectuées avant et après Dsatur, permettent de filtrer les 122 paires restantes. Le temps total sur cinq relances est 0.18 secondes. La solution est présentée dans la Figure 6.19.

9	4	1	2	**6**	**3**	**7**	5	**8**
8	3	**2**	7	5	4	9	1	**6**
5	6	**7**	1	**9**	8	3	4	2
4	**5**	8	**9**	1	2	6	**3**	7
6	**1**	3	**8**	7	**5**	2	**9**	4
2	**7**	9	**4**	3	**6**	5	**8**	1
7	2	4	3	**8**	9	**1**	6	5
1	9	5	6	4	7	**8**	2	**3**
3	8	**6**	**5**	**2**	1	4	7	9

Figure 6.19 – Résolution du problème SUDOKU de type moyen.

Une instance de problème très difficile de SUDOKU est présentée dans la Figure 6.20.

							3	
6			1					7
1		9	6					5
						5	6	1
				2	4			
	8							
						9		3
	4	5			2			
		8			7			

Figure 6.20 – Problème SUDOKU de type très difficile.

Cette instance comporte au départ 561 couples sommet-couleur à tester. Il y a 21 cases de remplies à l'origine. Le prétraitement effectué par la procédure `Réduction` au début de l'algorithme permet de marquer les 21 paires correspondant aux cases remplies au départ, de filtrer 302 paires et de supprimer 397 arêtes. Le tabou permet de marquer 60 paires. Les 178 paires restantes sont filtrées soit par Dsatur, soit par la procédure de réduction effectuée avant ou après Dsatur. Le temps total est 0.7 secondes. La solution est présentée dans la Figure 6.21.

En résumé, nous pouvons faire les remarques suivantes concernant les résultats obtenus pour ces instances provenant de problèmes SUDOKU. Premièrement, bien que ces instances aient aussi une seule solution comme celles présentées à la section 6.2.3, elles sont plus faciles à résoudre. Ceci est principalement dû au fait que certains sommets ont une seule couleur possible (= cases remplies au départ) et à cause de ceci, la procédure `Réduction` permet de filtrer de nombreuses paires impossibles (et même pour les instances les plus faciles, elle permet de filtrer toutes les paires impossibles) et de supprimer de nombreuses arêtes. Nous avons pu remarquer que pour les instances plus difficiles,

8	5	7	2	4	9	1	**3**	6
6	2	4	**1**	3	5	8	9	**7**
1	3	**9**	**6**	7	8	4	2	**5**
4	9	2	7	8	3	**5**	**6**	**1**
7	6	1	5	**2**	**4**	3	8	9
5	**8**	3	9	1	6	2	7	4
2	7	6	8	5	1	**9**	4	**3**
9	**4**	**5**	3	6	**2**	7	1	8
3	1	**8**	4	9	**7**	6	5	2

Figure 6.21 – Résolution du problème SUDOKU de type très difficile.

quand le prétraitement n'obtient pas l'unique solution, alors le tabou permet de marquer cette unique solution et Dsatur ainsi que la procédure de réduction effectuée avant ou après Dsatur permet de filtrer les paires restantes.

6.3 Conclusion

Ce chapitre a permis de présenter un algorithme de filtrage pour la contrainte `Some-Different` (i.e., pour le problème de coloration par listes) qui combine une recherche tabou afin de rapidement trouver un support pour le plus de couples sommet-couleur possible, et un algorithme exact afin de valider ou filtrer les couples sommet-couleur restants. Notre algorithme de filtrage s'est avéré être à peu près aussi rapide que celui de Richter et al. [122] quand il a été testé sur des données provenant d'un problème de planification de la main-d'oeuvre, et sensiblement plus rapide sur des données aléatoires.

Les principes généraux de l'approche proposée ne sont pas spécifiques à la sous-structure de coloration de graphe. En effet, cette technique peut être adaptée à d'autres contraintes NP-difficiles dans le but d'obtenir une procédure de filtrage qui impose la cohérence de domaine. Ceci peut être fait en développant principalement deux procédures spécifiques

de bas niveau (selon le problème) : une procédure de recherche locale pour trouver rapidement un support pour le plus de paires variables-valeurs et un algorithme exact pour valider ou filtrer les paires pour lesquelles la recherche locale n'a pas trouvé de support. Par conséquent, il serait intéressant dans le futur d'approfondir cette approche pour d'autres contraintes NP-difficiles.

CHAPITRE 7

REVUE DE LA LITTÉRATURE CONCERNANT LE PROBLÈME DE CONFECTION D'HORAIRES POUR LE PERSONNEL NAVIGANT AÉRIEN

Dans ce chapitre, nous décrivons en détail le problème de confection d'horaires pour le personnel navigant aérien et nous décrivons les méthodes existantes pour la résolution de ce problème. Comme nous l'avons vu brièvement dans le chapitre 2, ce problème peut être décrit à l'aide d'un CSP. Dans ce qui suit, nous allons d'abord décrire le contexte dans lequel se situe notre problème. Puis, nous présentons les méthodes de résolution existantes et pourquoi nous cherchons des IIS de contraintes dans ce problème.

7.1 Contexte

En transport aérien, il y a cinq étapes de planification.

1. Choix des destinations et des horaires pour chaque destination
2. Choix des avions pour chaque destination et construction des rotations d'avions
3. Construction des rotations de personnel
4. Construction des horaires mensuels
5. Modifications opérationnelles

Définition 7.1.1. Une *rotation* est une suite de vols et des temps de repos obligatoires s'y rattachant commençant et finissant à une base (aéroport où sont basés les employés). Par exemple, Toronto-Montréal, Montréal-Francfort, Francfort-Toronto est une rotation composée de trois vols.

L'étape 3 consiste à créer un ensemble de rotations de coût minimum qui respectent les conventions collectives de travail des employés et les lois gouvernementales et qui couvrent tous les vols de la compagnie.

L'étape numéro 4 de la planification consiste à créer un horaire pour chaque employé qui respecte lui aussi certaines règles (de la convention collective et de la sécurité aérienne). Des rotations et des temps de repos sont affectés à chaque employé pour une période d'un mois.

L'étape numéro 5 de la planification sert à tenir compte des changements qui peuvent survenir à tout moment (par exemple à cause d'un employé malade).

Dans cette thèse, nous allons intervenir au niveau de l'étape 4.

Il existe trois façons d'affecter du personnel à des horaires en transport aérien :

1. **Le bidline** : Premièrement, des blocs mensuels qui couvrent toutes les rotations sont construits (sans tenir compte des préférences individuelles du personnel). Ensuite, les employés indiquent leurs préférences concernant le bloc mensuel qu'ils aimeraient obtenir. Comme les blocs mensuels sont construits d'avance, ils ne tiennent pas compte des contraintes additionnelles de chaque employé comme les vacances ou les activités préaffectées (par exemple, les formations). Ceci engendre qu'il arrive très fréquemment qu'un employé reçoit un bloc mensuel en conflit avec ses activités prédéterminées. Les rotations en conflit doivent alors être réassignées manuellement à d'autres employés. Ceci est coûteux pour les compagnies aériennes et frustrant pour les employés.

2. **Le rostering** : Contrairement au bidline, aucun bloc mensuel n'est construit d'avance. Par contre, les employés indiquent leurs préférences en matière de vols

(destinations, heures des vols, etc.).
Ensuite un horaire mensuel est fait pour chaque employé. Ce système est basé sur le principe d'équité entre les employés. Cela veut dire qu'à long terme, on essaie d'obtenir le même taux de satisfaction pour tous les employés.
Ce système est principalement utilisé dans les compagnies aériennes européennes comme par exemple Air France [53], Alitalia [42,124], Lufthansa [60] ou Swissair [133].

3. **Le preferential bidding system, PBS** : Ce système est principalement employé en Amérique du nord où le principe de séniorité stricte s'applique. Il faut affecter les employés aux rotations selon les choix des employés de telle sorte que toutes les rotations soient effectuées et que chaque employé, selon l'ordre de séniorité, ait l'horaire qui le satisfait le plus et qu'aucun employé ne puisse améliorer sa situation sans détériorer celle d'un plus sénior.
C'est cette méthode d'affectation qui va nous intéresser.

Définition 7.1.2. Le terme de *tâche-mère* sera utilisé ici comme un synonyme de rotation. Nous allons utiliser la notation R pour désigner l'ensemble des rotations.

Définition 7.1.3. Une *tâche-fille* ou *tâche* est une copie d'une tâche-mère. Il y a une tâche-fille par employé requis pour effectuer la tâche-mère.

Définition 7.1.4. Une *qualification* est un attribut requis pour effectuer une tâche.
Des exemples de qualifications sont : parler le portugais, parler l'allemand, etc. Nous allons utiliser la notation Q pour désigner l'ensemble des qualifications.

Définition 7.1.5. Le *repos postcourrier* est une période de repos obligatoire suivant une tâche. Le temps de ce repos va être inclus dans le temps de la rotation pour la résolution du problème.

Définition 7.1.6. Une *préaffectation* est une tâche planifiée à l'avance qui doit être effectuée par un employé particulier.
Par exemple : les vacances, l'entraînement, la fin d'une tâche ayant débuté le mois précédent.

Définition 7.1.7. Les *crédits de vol,* w_t, sont le nombre d'heures ou de minutes qui sont créditées pour une tâche t à un employé p (les repos postcourrier et les temps morts entre les rotations ne sont pas comptés dans w_t).
Nous notons W_p la somme des crédits de vol w_t des tâches t effectuées par la personne p. Alors, pour chaque employé p, W_p doit être compris entre une valeur de crédit de vol minimale, L_p, et une valeur de crédit de vol maximale, U_p.

L'affection des tâches aux employés doit vérifier quatre types de contraintes.

1. **Contraintes de non-chevauchement** : Aucun employé ne peut effectuer plusieurs tâches simultanément.
2. **Contraintes dues aux qualifications** : Soit N_{qr} le nombre d'employés requis ayant la qualification q pour la rotation r. Il faut s'assurer que le nombre d'employés affectés à r et ayant la qualification q est supérieur ou égal à N_{qr}.
3. **Contraintes de crédits de vol** : Chaque employé doit accumuler W_p crédits de vols avec $L_p \leq W_p \leq U_p$.
4. **Contraintes de préaffectation** : Il faut tenir compte des préaffectations (congés, vacances, etc.).

7.2 Les méthodes de résolution du PBS

Dans la littérature, il y a moins d'articles sur les méthodes de résolution du PBS que sur celles concernant le rostering. Néanmoins, certaines compagnies ont proposé leurs solutions, comme par exemple Quantas [117], Midwest Express Airlines Inc. [125], CP

Air [29] ou Air Canada [54].

Supposons que nous ayons m employés à disposition et que l'ensemble des employés soit noté $E = \{1, \ldots, m\}$. Afin de résoudre le problème du PBS, il faut optimiser m différents problèmes successivement, un pour chaque employé, en partant du plus sénior et en allant vers le plus junior. Sherali et Soyster [130] ont formulé le PBS comme un problème en nombres entiers et utilisent une programmation multi-objectifs. Pour cela, ils cherchent différents poids $\{\lambda_1, \ldots, \lambda_m\}$ afin de donner une priorité différente aux différents objectifs. Mais Gamache et al. [54] ont montré que cette méthode nécessite l'utilisation de nombres qui sont trop grands pour les ordinateurs actuels.

Gamache et al.[54] ont présenté une méthode séquentielle combinant la génération de colonnes et un algorithme de séparation et d'évaluation progressive. Leur algorithme résout une suite de programmes linéaires mixtes : il y en un par employé $i \in E$ et les employés sont traités du plus sénior au plus junior. Leur algorithme consiste à construire le meilleur horaire possible pour l'employé k, en ne changeant rien aux horaires déjà déterminés pour les employés $1, \ldots, k-1$ et avec la contrainte qu'il doit être possible d'assigner un horaire aux employés plus juniors $k+1, \ldots, m$ de telle sorte que toutes les rotations de R soient couvertes. Nous allons utiliser les notations suivantes et nous allons supposer qu'un horaire admissible a déjà été assigné aux employés $1, \ldots, k-1$.

- $Q_{pq} = \begin{cases} 1 & \text{si l'employé } p \text{ a la qualification } q, \\ 0 & \text{sinon} \end{cases}$.
- N_{qr}^k est le nombre d'employés encore requis avec la qualification q dans la rotation r.
- b_r^k est le nombre d'employés encore requis par la rotation r.
- S_p^k est l'ensemble de tous les horaires admissibles restants pour l'employé p.
- $a_{rj} = \begin{cases} 1 & \text{si la rotation } r \text{ fait partie de l'horaire } j, \\ 0 & \text{sinon} \end{cases}$.

- c_{pj} est le score de l'horaire j pour l'employé p.
- $x_{pj} = \begin{cases} 1 & \text{si l'horaire } j \in S_p^k \text{ est choisi pour l'employé } p, \\ 0 & \text{sinon} \end{cases}$.

Ainsi, le problème consistant à trouver un horaire admissible pour l'employé k, tout en s'assurant que les employés plus juniors peuvent satisfaire les rotations restantes, peut être formulé comme le programme en nombres entiers (IP_k) suivant.

$$
(IP_k) \begin{cases}
\text{Max} \quad Z_{IP_k} = \displaystyle\sum_{j \in S_k^k} c_{kj} x_{kj} & & & \\
\text{s.c.} & & & \\
\displaystyle\sum_{p=k}^{m} \sum_{j \in S_p^k} a_{rj} Q_{pq} x_{pj} \geq N_{qr}^k, & \forall r \in R,\ \forall q \in Q & (7.1) \\
\displaystyle\sum_{p=k}^{m} \sum_{j \in S_p^k} a_{rj} x_{pj} = b_r^k, & \forall r \in R & (7.2) \\
\displaystyle\sum_{j \in S_p^k} x_{pj} = 1, & p = k, \ldots, m & (7.3) \\
x_{pj} \in \{0, 1\}, & p = k, \ldots, m & \\
& \forall j \in S_p^k & (7.4)
\end{cases}
$$

Les contraintes (7.1) concernent les qualifications. Les contraintes (7.2) s'assurent que les horaires choisis couvrent toutes les rotations et que le bon nombre d'employés est utilisé. Les contraintes (7.3) s'assurent qu'un horaire est construit pour chaque employé. Un algorithme séquentiel peut alors être utilisé qui détermine successivement un horaire pour chaque employé k en résolvant (IP_k). Cet algorithme produit une solution optimale

au PBS dans le cas où aucun employé n'a deux horaires ayant le même score optimal. En revanche, la résolution du problème en nombres entiers (IP_k) peut être longue et dans une solution optimale de (IP_k) les horaires pour les employés $k+1,\ldots,m$ sont inutiles car ils ne tiennent pas compte des préférences de ces employés. Afin d'essayer de tenir compte de ces deux remarques, Gamache et al. ont développés un programme linéaire mixte qui utilise des contraintes d'intégralité seulement pour les variables associées à l'employé k (et donc les variables associées aux employés $k+1,\ldots,m$ n'ont pas de contraintes d'intégralité). Ce programme linéaire mixte (MIP_k) est présenté ci-dessous.

$$
(MIP_k)\begin{cases}
\text{Max} \quad Z_{MIP_k} = \displaystyle\sum_{j\in S_k^k} c_{kj}x_{kj} & \\
\text{s.c} & \\
\displaystyle\sum_{p=k}^{m}\sum_{j\in S_p^k} a_{rj}Q_{pq}x_{pj} \geq N_{qr}^k, & \forall r\in R,\ \forall q\in Q \quad (7.5)\\
\displaystyle\sum_{p=k}^{m}\sum_{j\in S_p^k} a_{rj}x_{pj} = b_r^k, & \forall r\in R \quad (7.6)\\
\displaystyle\sum_{j\in S_p^k} x_{pj} = 1, & p=k,\ldots,m \quad (7.7)\\
x_{kj}\in\{0,1\}, & \forall j\in S_p^k \quad (7.8)\\
x_{pj}\geq 0, & p=k+1,\ldots,m \\
& \forall j\in S_p^k \quad (7.9)
\end{cases}
$$

Gamache et al. [53] résolvent le problème (MIP_k) en combinant un algorithme

de génération de colonnes avec un algorithme de séparation et évaluation progressive (branch-and-bound). La génération de colonnes sert à résoudre le programme linéaire qui est obtenu lorsque les contraintes d'intégrité de (MIP_k) sont relaxées. L'algorithme de séparation et évaluation progressive est utilisé pour trouver une solution entière pour l'employé k. Il arrive parfois que la méthode donne un horaire à un employé de telle sorte qu'il n'est plus possible de trouver un horaire réalisable pour les employés plus juniors. À ce moment-là, quand le (MIP_k) ne trouve pas de solution réalisable, la méthode doit effectuer un retour-arrière. Ce processus de retour-arrière est répété jusqu'à ce que, pour un employé $l < k$, le problème en nombres entiers (IP_l) ait une solution réalisable.

La génération de colonnes doit être faite pour chaque employé. Pour diminuer le temps de résolution, une heuristique est introduite durant la résolution des problèmes des premiers employés (au début de la résolution il y a beaucoup d'employés disponibles, donc l'affectation des tâches aux employés se fait relativement facilement). Cette méthode utilise un algorithme de plus court chemin et génère un horaire pour l'employé k sans tenir compte des employés plus juniors. Le problème résolu est appelé $(RCSP_k)$ (pour le terme anglais *resource constrained shortest path problem*) et est utilisé par Gamache et al. [54] pour résoudre les sous-problèmes dans la méthode de génération de colonnes.

$$(RCSP_k)\begin{cases} \text{Max} \quad Z_{RCSP_k} = \displaystyle\sum_{j \in S_k^k} c_{kj} x_{kj} \\ \text{s.c} \\ \qquad \displaystyle\sum_{j \in S_k^k} x_{kj} = 1, \\ \qquad x_{kj} \in \{0,1\}, \quad \forall j \in S_k^k \end{cases}$$

Comme nous l'avons déjà mentionné, la méthode (MIP_k) n'est pas optimale dans

le cas où un employé k a plusieurs horaires optimaux S_k^* qui garantissent que les employés juniors $k+1, \ldots, m$ reçoivent un horaire. Achour et al. [3] décrivent une méthode exacte qui ne présente pas ce problème. En effet, à la place de fixer un horaire optimal pour l'employé k, ils construisent l'ensemble noté $\widehat{\Omega}_k^k$ de tous les horaires résiduels admissibles pour l'employé k ayant le score S_k^* en énumérant tous ces horaires. Donc, le choix de l'horaire de l'employé k est reporté à une itération ultérieure, quand des informations supplémentaires sur les scores des employés plus juniors sont disponibles.

En résolvant successivement des problèmes $(RCSP_k)$ pour chaque employé selon l'ordre de séniorité, il est très probable que, pour un certain employé k, l'horaire obtenu par le $(RCSP_k)$ induise une solution non réalisable pour le problème (IP_k). Jeandroz [80] (pour les tâches-filles) et El Idrissi [41] (pour les tâches-mères) ont proposé d'utiliser des compteurs pour détecter quand le nombre d'employés plus juniors devient critique.

Cette méthode passe en revue les employés, selon l'ordre de séniorité.
Un problème de plus court chemin $(RCSP_k)$ est résolu pour l'employé le plus sénior k et on lui assigne temporairement cet horaire. Pour les employés résiduels $k+1, \ldots, m$ (qui n'ont pas encore un horaire assigné), des compteurs sont mis en place afin de vérifier s'il existe une solution admissible, tenant compte des contraintes du problème, sans prendre en considération les préférences de ces employés.
Si le résultat des compteurs le permet, l'horaire trouvé pour l'employé le plus sénior lui est attribué définitivement.
Sinon, les compteurs permettent de savoir quelles sont les qualifications critiques, et donc quelles tâches devront être assignées ou interdites à l'employé le plus sénior (sans tenir compte de ses préférences), afin de pouvoir assurer la réalisabilité future du problème.

En fait, il y a deux types de compteurs dans la procédure : les compteurs $C1$ et les

compteurs $C2$. Les compteurs $C1$ vérifient qu'il reste assez d'employés qualifiés dans le problème résiduel pour pouvoir effectuer les tâches demandant des qualifications. Les compteurs $C2$ vérifient que le nombre total de crédits de vol des tâches du problème résiduel est compris entre le total des nombres de crédits de vol minimum et le total des nombres de crédits de vol maximum des employés encore disponibles. Il y a aussi un compteur $C2$ *global* qui détermine le critère d'arrêt de l'heuristique.
Le problème est dit *critique* quand l'un des compteurs détermine que la demande est plus grande ou égale à l'offre.
Il y a un compteur $C1$ par qualification (allemand, portugais, etc.) et par tranche de temps (le temps est discrétisé à chaque heure de début ou de fin de tâche).
Un compteur $C1$ calcule l'offre et la demande pour une qualification et vérifie si le problème résiduel est réalisable dans un intervalle de temps donné.
Une qualification est critique si le retrait de la personne la plus sénior ne permet pas la couverture des tâches résiduelles nécessitant cette qualification. Dans ce cas-là, une tâche demandant cette qualification est attribuée à l'employé le plus sénior.

Les compteurs $C2$ indiquent quand l'utilisation des compteurs $C1$ doit cesser. Les compteurs $C2$ vérifient que le nombre de crédits de vol à assigner est compris entre la somme des valeurs de crédits de vol minimales et maximales des employés non encore assignés. Une fois que les compteurs $C2$ détectent que le problème devient critique, un retour à la génération de colonnes est fait.

Les compteurs peuvent ne pas détecter certaines situations où le problème est non réalisable, comme dans l'exemple suivant, où les compteurs considèrent que l'offre est 3 dans chaque intervalle de temps et que la demande est au maximum 2.

Exemple 7.2.1. Voici un exemple où il y a six rotations à effectuer, A, B, C, D, E, F, et

chaque rotation a une seule tâche-fille. Le tableau 7.1 donne les contraintes de qualifications à respecter pour chaque rotation ainsi que la durée des rotations.

Tableau 7.1 – Les qualifications requises pour les rotations ainsi que les durées des rotations

Rotation	Portugais	Italien	Français	Allemand	Durée
A	X				2
B		X			2
C		X			3
D			X		2
E				X	3
F				X	2

Il y a trois employés disponibles : 1, 2 et 3. Le tableau 7.2 donne les qualifications de chacun d'entre eux, ainsi que le nombre d'heures minimum et maximum qu'ils doivent effectuer.

Tableau 7.2 – Les qualifications des employés

Employé	Portugais	Italien	Français	Allemand	Min	Max
1	X	X			4	5
2		X		X	4	5
3			X	X	4	5

La figure 7.1 montre la disposition des rotations dans le temps.

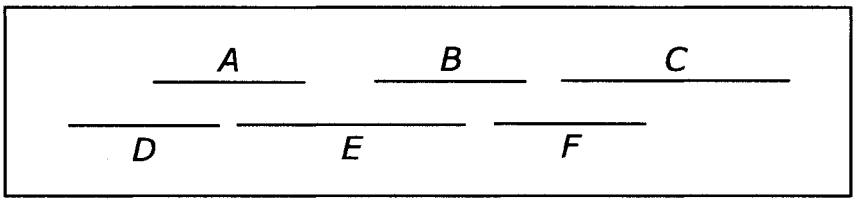


Figure 7.1 – Représentation des rotations selon leur apparition dans le temps

Notons x^p_{UV}, la variable indiquant que l'employé p effectue U puis V. Normalement, ce

sont des variables binaires. Mais si les contraintes d'intégrité sont relaxées, la méthode trouve la solution non entière suivante.

$$x^1_{AB} = x^1_{AC} = x^2_{BC} = x^2_{EF} = x^3_{DE} = x^3_{DF} = \frac{1}{2}$$

En effet, nous avons les contraintes suivantes sur les rotations (chaque rotation doit être effectuée totalement).

$$\sum_j x^p_j = 1 \ \forall p$$

Il est pourtant impossible de trouver une bonne affectation, car chaque personne doit faire exactement deux rotations pour que la charge de travail appartienne à $[4, 5]$.
Comme l'employé 1 doit faire A (car il est le seul à parler Portugais), nous savons qu'il fera B ou (exclusif) C (en effet, il ne peut pas faire F car il ne parle pas Allemand).
Comme l'employé 3 doit faire D (car il est le seul à parler Français), nous savons qu'il fera E ou (exclusif) F (en effet, il ne peut pas faire B ou C car il ne parle pas Italien).
Nous pouvons en déduire que l'employé 2 doit faire (B ou C) et (E ou F).
Si l'employé 2 effectue B, alors il ne peut pas faire E ou F car il y a un chevauchement des rotations.
Si l'employé 2 effectue C, alors il ne peut pas faire F, car il y chevauchement ; et il ne peut pas faire E car la durée totale des deux rotations vaut 6. Ainsi, nous pouvons en déduire qu'il est impossible de trouver une affectation qui vérifie toutes les contraintes.

Les problèmes (IP_k), (MIP_k) et $(RCSP_k)$ peuvent avoir des valeurs optimales différentes. C'est pour cela que quand un algorithme séquentiel est utilisé pour résoudre (MIP_k), il faut parfois effectuer des retours-arrières. Or, comme nous l'avons déjà vu, la résolution du problème (IP_k) peut être très longue. Afin d'essayer de trouver rapidement une solution au problème (IP_k), Gamache et al. [52] (provenant de la maîtrise de Jérôme Ouellet [110]) ont adapté la méthode de recherche tabou (présentée dans

l'Annexe I). Ils ont, en fait, généralisé l'algorithme TABUCOL proposé par Hertz et de Werra, [75], pour la k-coloration d'un graphe.
En fait, Gamache et al. [52] ont intégré leur recherche tabou dans un algorithme qui fonctionne de la façon suivante. Jusqu'à une certaine valeur de k fixée, appelée k_{max} ils trouvent d'abord un horaire pour l'employé k en résolvant le problème de plus court chemin $(RCSP_k)$. Puis, ils utilisent la recherche tabou pour déterminer si (IP_k) est réalisable. Si l'algorithme tabou exhibe une solution réalisable, alors cela signifie que l'horaire attribué à l'employé k est optimal. Sinon, si l'algorithme tabou n'est pas capable d'exhiber une solution à (IP_k) (i.e., l'output de tabou est "JE NE SAIS PAS"), alors ils trouvent un horaire pour k en résolvant le problème linéaire mixte (MIP_k) et testent l'horaire obtenu avec la recherche tabou. À nouveau, si tabou ne trouve pas de solution, alors ils résolvent le problème à nombres entiers (IP_k). Si $k > k_{max}$, alors ils passent directement à la résolution du problème mixte (MIP_k) sans essayer de résoudre d'abord le problème de plus court chemin.

Gamache et al. [52] ont modélisé le problème de confection d'horaires du personnel navigant comme un problème de D-coloration avec contraintes additionnelles.
Plus précisément, un graphe est construit, dans lequel

- chaque sommet est une tâche,
- il existe une arête entre deux sommets si et seulement si les deux tâches correspondantes se chevauchent,
- chaque couleur correspond à un employé,
- les domaines de chaque sommet sont composés des couleurs des employés pouvant effectuer la tâche associée au sommet.

S'il n'y avait que les contraintes de non-chevauchement, le problème à résoudre serait de déterminer une k-coloration d'un graphe d'intervalle (où k est le nombre d'employés). Ce problème peut être résolu en temps polynomial. Les préaffectations peuvent cependant imposer que deux sommets doivent avoir la même couleur et il a été démontré

que le problème devient alors NP-difficile [21], et l'utilisation d'une heuristique est donc justifiée.
Donc, en fait le CSP modélisant ce problème est constitué d'une contrainte SomeDifferent additionnée de contraintes de qualification et de contraintes de temps de travail minimum et maximum pour les employés.

Les contraintes de non-chevauchement sont modélisées par les arêtes qui imposent que deux sommets adjacents doivent avoir des couleurs différentes. Deux tâches qui se chevauchent ne peuvent donc pas être attribuées à un même employé.
Les contraintes de préaffectation sont modélisées en restreignant l'ensemble des couleurs (employés) pouvant être affectées à certains sommets (tâches).

Les contraintes dues aux qualifications sont modélisées comme suit.
Nous reprenons les notations de N_{qr} et Q_{pq} telles que définies ci dessus à la page 180.
De plus, soit $P_r(s)$ l'ensemble des personnes qui sont affectées à une tâche de la rotation r dans la solution s.

Alors les **contraintes dues aux qualifications** s'expriment de la façon suivante.

$$\sum_{p \in P_r(s)} Q_{pq} \geq N_{qr} \qquad \forall \text{ qualification } q \text{ requise dans une rotation } r$$

En termes de graphes cela donne la contrainte suivante.
Soit C_q l'ensemble des couleurs (employés) ayant la qualification q.
Soit V_r l'ensemble des sommets correspondant aux tâches de la rotation r.
Il faut qu'au moins N_{qr} sommets de V_r aient une couleur $C \in C_q$.

Les **contraintes de crédits de vol** sont modélisées comme suit.

Soit $W_p(s)$ le nombre de crédits de vols cumulés par l'employé p dans la solution s.

Soit L_p le nombre de crédits de vol minimum pour l'employé p.

Soit U_p le nombre de crédits de vol maximum pour l'employé p.

On a $L_p \leq W_p(s) \leq U_p$.

Associons un poids w_t à chaque tâche t qui correspond à sa durée (son nombre de crédits de vol). Notons $s(t) = p$ le fait que t a la couleur p dans la solution s.

Il faut que, pour toute couleur p, la contrainte suivante soit vérifiée :

$$W_p(s) = \sum_{s(t)=p} w_t \quad \in [L_p, U_p]$$

Si nous reprenons l'exemple présenté dans le Tableau 7.1, nous obtenons le graphe présenté dans la Figure 7.2. Chaque rotation est représentée par un sommet. Au-dessus de chaque sommet est indiqué le poids du sommet (qui représente la durée de la rotation).

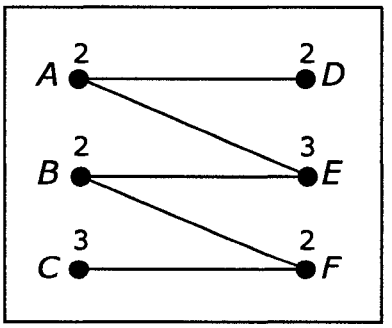


Figure 7.2 – L'exemple 7.2.1, présenté sous la forme d'un graphe

Nous cherchons une 3-coloration des sommets. Nous avons les contraintes supplé-

mentaires suivantes.

1. La somme des poids des sommets d'une même couleur appartient à $[4, 5]$.
2. A ne peut recevoir que la couleur 1.
3. B ne peut recevoir que les couleurs 1 ou 2.
4. C ne peut recevoir que les couleurs 1 ou 2.
5. D ne peut recevoir que la couleur 3.
6. E ne peut recevoir que les couleurs 2 ou 3.
7. F ne peut recevoir que les couleurs 2 ou 3.
8. E et C ne peuvent recevoir la même couleur car la somme de leurs poids vaut 6.

Si les arêtes (A, D) et (A, E) sont supprimées, le sous-graphe obtenu est toujours non réalisable.

L'algorithme tabou de Gamache et al. [52] colore d'abord gloutonnement le graphe en respectant les préaffectations. La fonction objectif est la somme pondérée de trois termes $f_1(s)$, $f_2(s)$ et $f_3(s)$.

1. $f_1(s)$ est le nombre d'arrêtes en conflit.
2. $f_2(s)$ comptabilise le nombre d'écarts par rapport à U_p et L_p,
   $f_2(s) = \sum_{p=k}^{m} \max\{0, L_p - W_p(s), W_p(s) - U_p\}$.
3. $f_3(s)$ comptabilise le nombre de problèmes de qualification,
   $f_3(s) = \sum_{qr} \max\{0, N_{qr}^k - \sum_{p \in P_r(s)} Q_{pq}\}$.

Lors de l'utilisation de l'algorithme tabou de Gamache et al., les employés sont traités selon la séniorité stricte.

D'abord, une solution pour l'employé le plus sénior est cherchée. Celui-ci est alors enlevé du problème (il y a donc une couleur en moins). Les tâches que l'employé le plus

sénior avait sont redonnées à d'autres employés de manière gloutonne.
Une solution sans conflit est cherchée. À ce moment-là, il arrive parfois que le problème n'a aucune solution.
C'est donc à cette étape-là que nous allons intervenir afin de trouver un sous-ensemble de contraintes rendant le problème impossible à résoudre.
En effet, un IIS représente une partie du problème qui donne une explication partielle de la non réalisabilité. Donc, la connaissance d'un IIS du problème peut être très utile dans le problème d'horaire de personnel navigant, car elle donne une idée concernant les données à changer (dans le problème original) pour obtenir un problème réalisable. Cela permettra de savoir quelles tâches devront être affectées à des employés spécifiques.

En résumé, le problème de confection d'horaires pour le personnel navigant aérien consiste à trouver un horaire pour chaque employé qui respecte un certain nombre de règles et de contraintes. Les contraintes que nous allons prendre en compte dans ce travail sont les contraintes de non-chevauchement, les contraintes de crédits de vol, les contraintes de préaffectation et les contraintes de crédits de vol minimum et maximum. Comme nous l'avons vu, Gamache et al. [52] ont proposé un algorithme tabou qui retourne pour chaque employé, soit "RÉALISABLE" avec une affectation qui satisfait toutes les contraintes, soit "JE NE SAIS PAS" quand il ne trouve pas une telle affectation. Quand l'algorithme tabou ne trouve pas de solution, il est très possible que le problème à cette étape-là ne soit pas réalisable, et donc il est alors très utile de trouver un IIS de contraintes expliquant cette non réalisabilité. C'est donc à cette étape-là du processus de confection d'horaires que nous allons intervenir en utilisant des méthodes de détection d'IIS pour ce problème. Les méthodes que nous avons développées sont présentées dans le Chapitre 8 et sont des adaptations des méthodes de Galinier et Hertz [48], présentées dans la Section 4.1.

7.3 Conclusion

Dans ce chapitre, nous avons présenté le problème de confection d'horaires pour le personnel navigant aérien ainsi qu'un survol des méthodes de résolution existantes pour ce problème.

Dans le chapitre 8, nous présentons une adaptation des algorithmes de détection automatique d'IIS de contraintes pour le problème de confection d'horaires pour le personnel navigant aérien, ainsi qu'un algorithme exact permettant de vérifier les sous-ensembles obtenus.

CHAPITRE 8

DÉTECTION DE SOUS-ENSEMBLES INCOHÉRENTS MINIMAUX DANS LE PROBLÈME DE CONFECTION D'HORAIRES POUR LE PERSONNEL NAVIGANT AÉRIEN

Dans ce chapitre, nous présentons les adaptations que nous avons faites pour utiliser les algorithmes `Insertion` et `HittingSet` présentés à la section 4.1 afin de trouver des IIS de contraintes pour le problème de confection d'horaires pour le personnel navigant aérien. Puis, nous présentons l'algorithme de recherche tabou qui est utilisé par les algorithmes `Insertion` et `HittingSet`. Ensuite, nous décrivons comment nous avons modifié l'algorithme exact `Dsatur` permettant de trouver le nombre chromatique d'un graphe afin qu'il puisse tenir compte des contraintes additionnelles de qualification et de crédits de vol minimum et maximum et qu'il retourne la réponse réalisable ou non réalisable. Cet algorithme modifié nous permettra de vérifier les résultats des algorithmes de recherche d'IIS. Finalement, nous présentons divers résultats expérimentaux.

8.1 Algorithmes de détection de sous-ensembles incohérents minimaux

Dans cette section, nous présentons comment les algorithmes `Insertion` et `HittingSet` ont été adaptés pour trouver des IIS de contraintes pour le problème de confection d'horaires pour le personnel navigant aérien.

Tout d'abord, nous allons brièvement rappeler comment nous modélisons ce problème. Nous utilisons un graphe $G = (V, E)$ où V est l'ensemble des sommets et E est l'ensemble des arêtes. Chaque sommet $v \in V$ représente une tâche (tâche-fille) à effectuer et chaque arête $(u, v) \in E$ représente la contrainte de non-chevauchement entre deux tâches ayant lieu en même temps. Chaque sommet a une liste de couleurs possibles no-

tée D_v, chaque couleur représentant un employé pouvant effectuer la tâche associée au sommet. Il y a au total m employés, donc m couleurs.

Pour chaque contrainte de qualification, il faut s'assurer que le nombre d'employés affectés à la rotation r (ensembles de tâches) et ayant la qualification q soit supérieur ou égal à N_{qr}. Nous allons noter CQ_{qr} cette contrainte de qualification demandant N_{qr} personnes avec la qualification q dans la rotation r. Chaque couleur (= employé) doit effectuer un nombre de crédits de vol W_p compris entre L_p (nombre de crédits de vol minimum) et U_p (nombre de crédits de vol maximum).

De plus, rappelons les notations suivantes :

$$Q_{pq} = \begin{cases} 1 & \text{si } p \text{ a la qualification } q \\ 0 & \text{sinon} \end{cases}$$

$P_r(c)$ est formé par l'ensemble des personnes qui sont affectées à une tâche de la rotation r dans une solution c. Les contraintes de préaffectations contraignent certains domaines D_v à être des singletons (une seule personne peut être affectée à la tâche v).

Pour ce problème de confection d'horaires, nous allons utiliser les algorithmes `Insertion` et `HittingSet` et nous allons uniquement rechercher des IIS de contraintes. L'algorithme `Insertion` a été présenté au chapitre 4 dans la Figure 4.4 et l'algorithme `HittingSet` dans la Figure 4.8. Nous précisons maintenant les termes utilisés par ces deux algorithmes pour le cas particulier du problème de confection d'horaires pour le personnel navigant aérien. L'ensemble S utilisé par ces deux algorithmes est l'ensemble de toutes les contraintes du problème. Rappelons que $f_S(e)$ est le nombre de contraintes de S violées par une affectation e et $F_S(e)$ est formé par l'ensemble de ces contraintes de S violées par e. Nous avions vu que, dans le cas du problème SAT, la procédure `MIN` résolvait un problème Max-SAT pondéré. Dans le cas qui nous intéresse maintenant, la procédure `MIN` est très similaire. C'est une procédure qui prend en entrée un problème tel que décrit ci-dessus et l'ensemble des poids des contraintes (au départ toutes les contraintes ont un poids de 1) et telle que `MIN`(S_1, S_2) produit une affectation

(coloration) e des sommets qui minimise $\alpha \cdot f_{S_1}(e) + f_{S_2}(e)$ (où $\alpha > |S_2|$). Donc, les contraintes de S_1 reçoivent un poids de α tandis que les contraintes de S_2 ont un poids de 1. En pratique, nous utilisons une version heuristique de `MIN` : `HMIN`. À nouveau, nous employons l'algorithme de recherche tabou pour résoudre `HMIN`, i.e., pour trouver une affectation e des variables qui essaie de satisfaire toutes les contraintes de S_1 car elles ont un poids α et qui essaie de minimiser la somme des contraintes violées de S_2 (car elles ont un poids de 1). Les détails de cet algorithme de recherche tabou sont précisés dans la section suivante.

L'algorithme `HittingSet` utilise la procédure `HS` qui retourne un hitting set minimum. Comme pour le problème SAT, en pratique, nous trouvons un hitting set minimum en résolvant un programme linéaire en nombre entier (résolu avec CPLEX 8.1) comme décrit dans la section 4.4.

Les propriétés énoncées dans la section 4.1 sont toujours valides. Rappelons que celles-ci précisent que si le sous-ensemble obtenu comme sortie de l'algorithme `HInsertion` ou `HHittingSet` (i.e., utilisant la version heuristique de `MIN`, `HMIN`) est non réalisable, alors c'est un IIS (Galinier et Hertz [48]).

Dans les expériences que nous avons effectuées et qui sont présentées à la section 8.4, nous utilisons aussi la version heuristique de l'algorithme `PreFiltering` présenté à la section 4.2.1. Pour cela nous utilisons la même procédure `HMIN` que pour l'algorithme `Insertion` telle que nous allons la décrire ci-dessous.

8.2 Algorithme tabou

Comme nous l'avons mentionné, nous utilisons une heuristique de recherche tabou pour résoudre le problème `HMIN`. En effet, bien que nous allons décrire dans la section 8.3 un algorithme exact de résolution du problème de confection d'horaires pour le personnel navigant aérien, celui-ci n'est efficace (en termes de temps de calcul) en pratique que sur des petits problèmes. Donc il ne nous est utile que pour vérifier les IIS-C que

les algorithmes de détection vont exhiber, pour autant que ces IIS-C soient de taille pas trop grande. Mais l'algorithme exact serait trop lent pour être utilisé à l'intérieur des algorithmes `Insertion` ou `HittingSet`.

L'algorithme de recherche tabou utilisé pour résoudre `HMIN` est présenté dans la Figure 8.1. Cet algorithme est très similaire à celui que nous avons utilisé pour résoudre le problème SAT et aussi très proche de l'algorithme tabou de Gamache et al. [52]. Nous représentons l'espace des solutions au moyen de la lettre $\mathcal{S}$: nous employons cette notation, car c'est celle qui est principalement utilisée dans la littérature pour les méthodes de recherche tabou. L'espace des solutions $\mathcal{S}$ est formé par l'ensemble de toutes les colorations complètes possibles des sommets notées $c : V \rightarrow D(V)$ avec $c(v) \in D_v$ pour tout $v \in V$. Une solution voisine $c' \in N(c)$ est obtenue en donnant une couleur différente $c'(v) \neq c(v)$ à un sommet v appartenant à une contrainte en conflit.
L'ensemble des contraintes est noté C et une contrainte $C \in \mathcal{C}$. L'ensemble des poids des contraintes est notée Ω et chaque contrainte C a un poids $\omega(C)$. Comme nous l'avons vu, les poids des contraintes sont modifiés par l'algorithme `Insertion` ou `HittingSet`.

Nous précisons d'abord les termes qui sont utilisés dans le calcul de la fonction objectif.
Afin de pouvoir décrire la participation à la fonction objectif de chaque type de contrainte, nous introduisons les notations K et κ. La violation de chaque type de contrainte est pondérée par la valeur K plus la valeur de la violation. En pratique, la valeur de K va dépendre des instances testées. Pour les premières instances testées, nous avons fixé $K = 100$ et pour le deuxième groupe d'instances testées nous avons fixé $K = 10000$. Le saut de K ajouté à chaque contrainte permet de différencier un mouvement permettant de passer d'une contrainte violée à une contrainte non violée (ou l'inverse) d'un mouvement passant d'une contrainte violée à une contrainte violée mais moins (ou plus). Les contraintes de qualification sont pondérées par κ. Ceci a été fait afin que les contraintes

Tabou($\mathcal{F}, \Omega$)

Entrée : Un problème $\mathcal{F}$ supposément non réalisable et l'ensemble Ω des poids des contraintes.

Sortie : Une coloration c^* et un sous-ensemble U de contraintes non satisfaites ou une coloration c^* prouvant que l'instance de départ était réalisable.

Initialisation

pour chaque $v \in V$ **faire**
  $c(v) \leftarrow$ attribution aléatoire d'une couleur appartenant à D_v;
  $\Lambda(v, i) \leftarrow 0 \; \forall \; i \in D_v$;
$c^* \leftarrow c$ et $iter \leftarrow 1$ et $U \leftarrow \emptyset$;
pour chaque $C \in \mathcal{C}$ **faire**
  si *C n'est pas satisfaite* **alors** $U \leftarrow U \cup \{C\}$;

Tabou

tant que *aucun critère d'arrêt n'est rencontré* **faire**
  $ListeMvt \leftarrow \emptyset$ et $f_{best} \leftarrow +\infty$;
  pour chaque *sommet v appartenant à une contrainte de U* **faire**
    pour chaque *couleur* $i \in D_v \backslash \{c(v)\}$ **faire**
      $c'(v) \leftarrow i$;
      si $iter > \Lambda(v, c'(v))$ **ou** $f_\omega(c') < f_\omega(c^*)$ **alors**
        si $f_\omega(c') < f_{best}$ **ou** $ListeMvt = \emptyset$ **alors**
          $ListeMvt \leftarrow \{(v, c'(v))\}$;
          $f_{best} \leftarrow f_w(c')$;
        sinon si $f_\omega(c') = f_{best}$ **alors**
          $ListeMvt \leftarrow ListeMvt \cup \{(v, c'(v))\}$;
  si $ListeMvt = \emptyset$ **alors**
    $v' \leftarrow$ choisir au hasard une variable v appartenant à une contrainte de U;
    $c'(v') \leftarrow$ choisir au hasard une couleur appartenant à $D_{v'}$ différente de $c(v')$;
  sinon
    $(v', c'(v')) \leftarrow$ choisir au hasard un mouvement appartenant à $ListeMvt$;
  $c(v') \leftarrow c'(v')$;
  $\Lambda(v', c(v')) \leftarrow iter + \tau$;
  si $f_\omega(c) < f_\omega(c^*)$ **alors** $c^* \leftarrow c$;
  `MettreAJour` $(U, \mathcal{C}, (v', c(v'))$;
  $iter \leftarrow iter + 1$;
retourner U;

Figure 8.1 – Algorithme tabou pour le problème de confection d'horaires pour le personnel navigant aérien.

MettreAJour(U,$\mathcal{C}$,(v', $c(v')$))

Entrée : L'ensemble U des contraintes non satisfaites, l'ensemble $\mathcal{C}$ de toutes les contraintes et le sommet changé v' ainsi que sa nouvelle couleur $c(v')$.

Sortie : L'ensemble U mis à jour.

pour chaque $C \in \mathcal{C}$ *contenant* v' **faire**
  si $C \in U$ *et* C *est satisfaite en donnant la couleur* $c(v')$ *à* v' **alors**
    $U \leftarrow U \setminus \{C\}$;
  sinon si $C \notin U$ *et* C *est violée en donnant la couleur* $c(v')$ *à* v' **alors**
    $U \leftarrow U \cup \{C\}$;

Figure 8.2 – Procédure mettant à jour U.

de qualification ne soient pas favorisées par rapport aux contraintes de crédits de vol minimum et maximum. En pratique, nous prenons κ comme étant égale au nombre de crédits moyens d'une tâche.

Soit $f_a(c)$ la fonction pénalisant les violations sur les arêtes, $f_a(c)$ est calculée de la façon suivante.

$$f_a(c) = \begin{cases} K & \text{si l'arête } a \text{ est en conflit dans la coloration } c \\ 0 & \text{sinon.} \end{cases}$$

Soit $f_{CQ_{qr}}(c)$ la fonction pénalisant les violations sur les contraintes de qualification CQ_{qr}. Alors $f_{CQ_{qr}}(c)$ est calculée de la façon suivante.

$$f_{CQ_{qr}}(c) = \begin{cases} 0 & \text{si } \sum_{p \in P_r(c)} Q_{pq} \geq N_{qr} \\ K + \kappa \cdot (N_{qr} - \sum_{p \in P_r(c)} Q_{pq}) & \text{sinon} \end{cases}$$

Soit $f_{MIN_p}(c)$ la fonction calculant les pénalités par rapport au nombre de crédits manquants pour obtenir le nombre minimum de crédits de vol et $f_{MAX_p}(c)$ la fonction calculant les pénalités par rapport au nombre de crédits en trop par rapport au nombre maximum de crédits de vol.

Les termes $f_{MIN_p}(c)$ et $f_{MAX_p}(c)$ sont calculés de la façon suivante.

$$f_{MIN_p}(c) = \begin{cases} 0 & \text{si } W_p \geq L_p \\ K + (L_p - W_p) & \text{si } W_p < L_p \end{cases}$$

$$f_{MAX_p}(c) = \begin{cases} 0 & \text{si } W_p \leq U_p \\ K + (W_p - U_p) & \text{si } W_p > U_p \end{cases}$$

Nous pouvons remarquer que la valeur de la violation est prise en compte dans les termes concernant les contraintes de qualification, de crédits de vol minimum et maximum. En effet, si nous ne pondérons pas les termes concernant la violation de ces contraintes, alors nous ne tenons pas compte de l'éloignement par rapport au but de la contrainte. Par exemple, si le nombre de crédits de vol minimum est 3900 considérons les deux cas suivants :

1. le total actuel des crédits d'une couleur est 3899,
2. le total actuel des crédits d'une couleur est 100.

Il est clairement évident que le cas numéro 2 viole la contrainte plus fortement que le cas numéro 1 et il est a priori plus facile d'effectuer un mouvement ayant pour but de satisfaire la contrainte de crédits de vol minimum dans le cas 1 que dans le cas 2. C'est pour tenir compte de cette différence que les contraintes de qualification, de crédits de vol minimum et de crédits de vol maximum ont été pondérées par ce qu'il manque pour satisfaire la contrainte.

Nous notons les poids dans la solution actuelle de la façon suivante : ω_a est le poids de l'arête a, $\omega_{CQ_{qr}}$ est le poids de la contrainte de qualification CQ_{qr}, ω_{MIN_p} est le poids de la contrainte de crédits de vol minimum pour la personne p et ω_{MAX_p} est le poids de la contrainte de crédits de vol maximum pour la personne p. La fonction objectif $f_\omega(c)$ est la suivante :

$$f_\omega(c) = \sum_a f_a(c)\omega_a + \sum_{q,r} f_{CQ_{qr}}(c)\omega_{CQ_{qr}} + \sum_{p=1}^{m} f_{MIN_p}(c)\omega_{MIN_p} + \sum_{p=1}^{m} f_{MAX_p}(c)\omega_{MAX_p}.$$

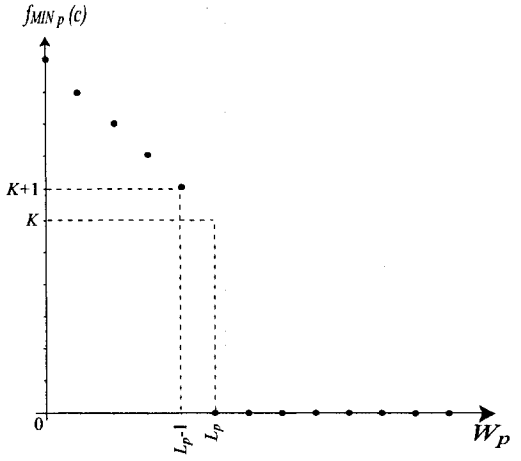
Nous pouvons voir dans la Figure 8.3 les allures des trois termes suivants appartenant à

la fonction objectif : $f_{MIN_p}(c)$, $f_{MAX_p}(c)$ et $f_{CQ_{qr}}(c)$.

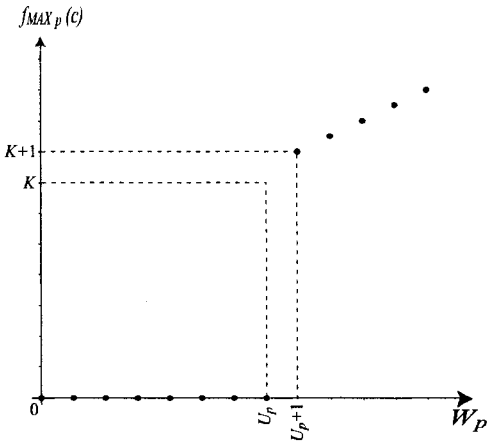
Si une instance ne comporte pas tous les types de contraintes, alors $f_\omega(c)$ pour cette instance comporte uniquement les parties qui concernent des types de contraintes qui composent l'instance.

Nous pouvons remarquer que le terme de la fonction objectif concernant chaque contrainte est pondéré par les termes ω_a, $\omega_{CQ_{qr}}$, ω_{MIN_p} ou ω_{MAX_p} selon le type de contrainte. Cette pondération varie en fonction de l'appartenance de la contrainte aux ensembles S_1 ou S_2 dans les algorithmes `Insertion` et `HittingSet` tel que décrit ci-dessus.

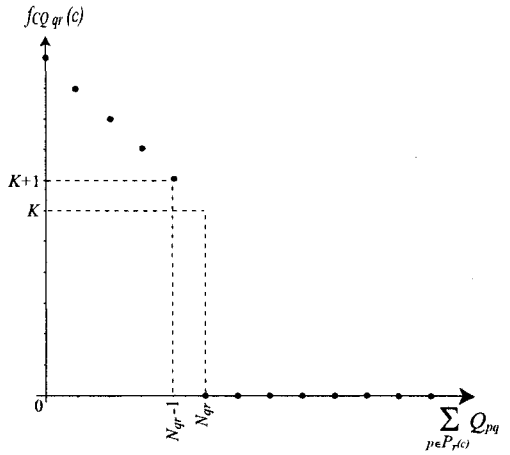
En pratique, le calcul de la fonction objectif n'est pas refait pour évaluer tous les mouvements. En effet, afin de pouvoir effectuer un mouvement avec un temps de calcul constant, nous maintenons à jour une matrice appelée $Gamma$ de taille $|V| \times m$ (i.e., nombre de sommets par nombre de couleurs) telle que $Gamma(v, i)$ contient le coût de l'affectation de la couleur i à v. Si la solution actuelle pour le sommet v est $c(v)$ et que nous voulons connaître le coût (positif ou négatif) du mouvement colorant v avec $c'(v)$ il nous suffit de calculer la différence entre le coût d'affecter $c'(v)$ à v et le coût de lui laisser sa couleur courante $c(v)$ (i.e., il faut effectuer le calcul $Gamma(v, c'(v)) - Gamma(v, c(v))$). Quand un mouvement est effectué il faut mettre à jour la matrice. Cette mise à jour doit être faite pour chaque type de contrainte touchant le sommet v. Pour les contraintes de non chevauchement, il faut mettre à jour les lignes de $Gamma$ concernant les sommets w voisins de v. Pour chacune de ces lignes, uniquement les colonnes concernant $c(v)$ et $c'(v)$ sont modifiées de la façon suivante. Premièrement, le coût d'affecter l'ancienne couleur de v, $c(v)$, est enlevé de w (i.e., le coût de la case $(w, c(v))$ est diminué) puis le coût d'affecter la nouvelle couleur de v, $c'(v)$ est ajouté à w (i.e., le coût de la case $(w, c'(v))$ est augmenté). Pour les contraintes de qualification et de crédits de vol minimum et maximum, les mises à jour sont un peu plus compliquées.



(a) Graphique de la fonction $f_{MIN_p}(c)$.



(b) Graphique de la fonction $f_{MAX_p}(c)$.



(c) Graphique de la fonction $f_{CQ_{qr}}(c)$ si $\kappa = 1$.

Figure 8.3 – Graphiques de fonctions pondérées.

Sans entrer trop dans les détails, l'idée générale est la suivante. Nous supposons que le mouvement consiste à changer la couleur du sommet v en passant de $c(v)$ à $c'(v)$. Pour chaque contrainte de qualification CQ_{qr} dont le nombre actuel de personnes ayant la qualification change à cause de ce mouvement (i.e., $\sum_{p \in P_r(c)} Q_{qp}$ est modifié), alors il faut mettre à jour les lignes de $Gamma$ correspondant aux sommets appartenant à la rotation r de la contrainte CQ_{qr}. Pour chaque contrainte de crédits de vol minimum ou maximum, il faut mettre à jour toutes les lignes de $Gamma$ correspondant à des sommets ayant soit la couleur $c(v)$ ou la couleur $c'(v)$ dans leur domaine.

Quand un mouvement est effectué pour passer d'une solution c à une solution c' en affectant une couleur $c'(v)$ à un sommet $v \in V$, la paire $(v, c(v))$ est rendue tabou. Il est donc interdit de réaffecter la couleur $c(v)$ à v pendant un nombre d'itérations τ fixé. Ceci a pour but de sortir des minima locaux et d'éviter de cycler. Comme nous l'avons déjà vu pour le tabou utilisé pour `MawWSAT`, la valeur de τ peut être calculée de différentes façons. Dans les tests que nous avons effectués, τ est choisi au hasard dans l'intervalle $[\alpha\sqrt{NV}, 2\alpha\sqrt{NV} - 1]$ où α est un paramètre et NV est le nombre de contraintes violées. Nous avons utilisé $\alpha = 1.5$ dans les tests que nous avons réalisés. Pour savoir si un mouvement est tabou $\Lambda(v, c(v))$ stocke le numéro de l'itération à partir duquel le mouvement $(v, c(v))$ n'est plus tabou. Chaque fois qu'un mouvement $(v, c(v))$ est effectué, la valeur de $\Lambda(v, c(v))$ est mise à jour et vaut le numéro de l'itération courante plus τ.
La valeur retournée par l'algorithme tabou est une coloration complète c^* des sommets de V et un ensemble U de contraintes non satisfaites par c^* tel que $f_\omega(c^*)$ soit la plus petite valeur de la fonction objectif rencontrée si nous n'avons pas pu prouver que l'instance est réalisable ou une coloration complète c^* des sommets de V qui satisfait toutes les contraintes sinon.
L'algorithme s'arrête soit après un nombre maximum fixé d'itérations ou d'itérations sans améliorations ou après un temps maximum d'exécution.
Dans la Figure 8.2 est présentée la procédure mettant à jour la liste U des contraintes non

satisfaites. Chaque contrainte qui était satisfaite avant d'effectuer le mouvement et qui devient non satisfaite par le mouvement est ajoutée à U et chaque contrainte qui appartenait à U avant le mouvement et qui devient satisfaite par le mouvement est supprimée de U.

8.3 Algorithme exact de vérification des sous-problèmes incohérents

Afin de pouvoir vérifier si un problème ou un sous-problème supposé non réalisable l'est effectivement, nous avons modifié l'algorithme exact `Dsatur` [115] de Michael Trick [134] afin qu'il tienne compte des contraintes de qualification et de crédits de vol minimum et maximum. Comme nous l'avons déjà vu au Chapitre 6, l'algorithme `Dsatur` calcule le nombre chromatique $\chi(G)$ d'un graphe G pour le problème classique de coloration de graphe. Afin d'adapter cet algorithme pour notre problème de confection d'horaires, nous avons dû effectuer certaines modifications. Premièrement, nous devons transformer notre problème de D-coloration en un problème classique de coloration de graphe. Pour faire cela, nous appliquons la même transformation que celle présentée dans la Section 6.1.3, i.e., nous ajoutons une clique de la taille $|D(V)|$ où chaque sommet correspond à une couleur de $D(V)$ et nous relions chaque sommet $v \in V$ à chaque nouveau sommet i si et seulement si $i \notin D_v$.

Dans les algorithmes, nous utilisons les notations suivantes. Soit $n = |V|$ le nombre de sommets du problème de D-coloration (=problème original) et $|D(V)|$ le nombre de couleurs utilisées (= la taille de clique ajoutée), alors $N = n + |D(V)|$ est le nombre de sommets du graphe modifié (utilisable pour le problème classique de coloration). L est la liste des sommets manipulés. Le nombre total de contraintes de l'instance ou de l'IIS testé est noté nc_{tot}. Ce nombre total est la somme du nombre de contraintes de non chevauchement, noté nc_a, du nombre de contraintes de qualification, noté nc_q, du nombre de contraintes de crédits de vol minimum, noté nc_{min}, et du nombre de contraintes de crédits de vol maximum, noté nc_{max}. Ainsi, $nc_{tot} = nc_a + nc_q + nc_{min} + nc_{max}$.

Deuxièmement, nous ajoutons des vérifications pour les contraintes de qualification. Pour cela, nous définissons $CtrL_v$ la liste pour chaque sommet v des contraintes de qualification dans lesquelles v est impliqué. Nous notons $Reserve_{CQ_{qr}}$ une variable stockant pour chaque contrainte de qualification CQ_{qr} la marge dont nous disposons concernant le nombre de sommets du graphe (rappelons que les sommets correspondent aux tâches) qui peuvent encore obtenir une couleur ayant la qualification demandée. Au début de l'algorithme, $Reserve_{CQ_{qr}}$ est initialisé avec le nombre de sommets modélisant la rotation r (i.e., le nombre de tâches (tâches-filles) de la rotation r) moins N_{qr} qui est le nombre minimum demandé d'employés ayant la qualification q pour la rotation r. Quand l'algorithme essaie d'affecter une couleur p au sommet v (Figure 8.4), pour toutes les contraintes $CQ_{qr} \in CtrL_v$, si la couleur p n'est pas qualifiée pour CQ_{qr}, $Reserve_{CQ_{qr}}$ est mis à jour et diminué de un (car il y a un sommet de la rotation r de moins qui peut obtenir une couleur ayant la qualification demandée). Si $Reserve_{CQ_{qr}}$ devient négatif, alors nous pouvons effectuer un retour-arrière, car la contrainte de qualification CQ_{qr} ne peut pas être satisfaite avec l'affectation partielle actuelle (il ne reste plus assez de sommets de r qui peuvent obtenir une couleur qualifiée). Quand une couleur p est enlevée d'un sommet v le mouvement opposé est effectué (Figure 8.5).

Troisièmement, nous ajoutons des vérifications pour les contraintes de crédits de vol minimum et maximum. Soit $Credits_v$ le nombre de crédits de vol du sommet v et L_p (respectivement U_p) le nombre minimum (respectivement maximum) de crédits de vols qui doivent être effectués par la couleur p (= personne p). Nous considérons aussi une valeur $CreditsActuels_p$ qui enregistre le nombre de crédits de vol actuels attribués à la couleur p (= effectués par la personne p). Comme nous l'avons précisé ci-dessus, nc_{min} est le nombre de contraintes de crédits de vol minimum de l'instance ou de l'IIS considéré (ce nombre n'est pas forcément égal au nombre de couleurs pour un IIS car un IIS peut ne contenir aucune contrainte de crédit de vol minimum ou moins de contraintes de

crédits de vol minimum que le nombre de couleurs).

Avant d'expliquer comment nous pouvons accélérer les vérifications pour les contraintes de crédits de vol minimum, nous considérons l'exemple suivant afin d'en avoir l'intuition.

Supposons que nous ayons x tâches à effectuer, chacune étant de durée 1 (avec x impair). Supposons que nous ayons deux personnes à disposition (donc deux couleurs), A et B, et que le nombre de crédits de vol minimum pour ces deux personnes soit $L_A = L_B = \frac{x+1}{2}$. C'est seulement après avoir affecté un ensemble de $\frac{x+1}{2}$ tâches à A (qui donnent un crédit total de L_A à A) que nous remarquons qu'il reste $\frac{x-1}{2} < L_B$ crédits disponibles pour B. Chacun des $C^x_{L_A} = \binom{x}{L_A}$ sous-ensembles de tâches aboutissent à ce cas-là. L'algorithme va donc tester $C^x_{L_A}$ solutions partielles avant de déterminer que le problème est irréalisable.

Par contre, si nous considérons l'ensemble de sous-ensembles de couleurs suivant $EP = \{\{A\}, \{B\}, \{A, B\}\}$ et que nous testons les contraintes de crédits de vol sur chacun de ces sous-ensembles alors nous remarquons immédiatement (sans explorer aucune solution partielle) que le problème est non réalisable. En effet, la somme du nombre de crédits de chaque tâche est x et cette somme est strictement plus petite que la somme du nombre minimum de crédit pour A et B : $x < L_A + L_B = x + 1$ donc ces deux contraintes de crédits de vol minimum ne peuvent pas être satisfaites simultanément.

Ainsi, afin d'accélérer les vérifications pour les contraintes de crédits de vol minimum, nous allons considérer un ensemble EP de sous-ensembles de couleurs pour lesquelles il y a une contrainte de crédit de vol minimum. Comme le nombre de tels sous-ensembles peut exploser selon la valeur de nc_{min}, nous ne les énumérons pas forcément tous. En fait, si nc_{min} est plus petit ou égal à 10, tous les sous-ensembles de couleurs de taille 1 à nc_{min} sont considérés, il y a alors $2^{nc_{min}} - 1$ sous-ensembles de couleurs ; sinon seulement les sous-ensembles de tailles 1, 2, 3 et nc_{min} sont utilisés.

Par exemple, si $nc_{min} = 4$ et que les couleurs pour lesquelles il y a une contrainte de crédit de vol minimum sont 1, 2, 3 et 4, alors $EP = \{\{1\}, \{2\}, \{3\}, \{4\}, \{1, 2\}, \{1, 3\},$

$\{1,4\},\{2,3\},\{2,4\},\{3,4\},\{1,2,3\},\{1,2,4\},\{1,3,4\},\{2,3,4\},\{1,2,3,4\}\}$.
Afin d'énumérer efficacement tous les sous-ensembles d'un ensemble, nous utilisons l'algorithme décrit par Loughry et al. [97]. Nous notons $ep \in EP$ un sous-ensemble de couleurs dans EP. Pour un ensemble $ep \in EP$, $Reste_{ep}$ donne la charge totale des tâches/sommets qui n'ont pas encore de couleur affectée et qui peuvent recevoir au moins une couleur/personne de ep. La matrice $ColorAdj_{vp}$ donne le nombre de voisins du sommet v qui ont la couleur p.

La procédure d'affectation d'une couleur à un sommet est présentée dans la Figure 8.4.

Quand nous affectons une couleur p à un sommet v nous mettons à jour $CreditsActuels_p$ en ajoutant le nombre de crédits du sommet v : $Credits_v$. S'il y a une contrainte de crédit de vol maximum pour la couleur p et si le nombre de crédits de vol actuellement attribués à la personne p est strictement plus grand que le nombre maximum de crédits de vol autorisés (i.e., si $CreditsActuels_p > U_p$) nous devons effectuer un retour-arrière car la contrainte n'est plus satisfaite et cela ne sert à rien d'évaluer les branches inférieures de l'arbre.
Pour les contraintes de crédits de vol minimum, les vérifications sont un peu plus compliquées.

- Premièrement, pour tous les ensembles ep qui contiennent au moins une couleur $j \in ep$ qui n'est attribuée à aucun voisin de v (i.e., telle que $ColorAdj_{vj} = 0$) nous mettons à jour $Reste_{ep}$ en lui soustrayant le nombre de crédits de vol de la tâche v (i.e., $Credits_v$) car cela signifie que la tâche v ne peut plus contribuer à $Reste_{ep}$ alors que sa contribution était de $Credits_v$ avant l'affectation.
- Deuxièmement, pour tous les voisins u de v nous mettons à jour la variable $ColorAdj_{up}$ stockant le nombre de voisins de couleur p de u en lui additionnant un (car v vient d'obtenir la couleur p). Si u n'est pas coloré, et de plus, si u a maintenant un seul voisin de couleur p (i.e., $ColorAdj_{up} = 1$ après mise à jour, donc v est le seul voisin de couleur p de u), alors pour tous les sous-ensembles ep conte-

`AffecterCouleur`(v, p)

Entrée : Un sommet v et une couleur p

Sortie : $VRAI$ s'il est possible d'affecter p à v en tenant compte des contraintes de qualification et des contraintes de crédit de vol MIN et MAX, $FAUX$ sinon.

OK $\leftarrow$ VRAI.

Affecter p à v.

$CreditsActuels_p \leftarrow CreditsActuels_p + Credits_v$.

pour chaque *ensemble ep contenant au moins* $j \in ep$ *tel que* $ColorAdj_{vj} = 0$ **faire**

  $Reste_{ep} \leftarrow Reste_{ep} - Credits_v$.

pour chaque *sommet u adjacent à v* **faire**

  $ColorAdj_{up} \leftarrow ColorAdj_{up} + 1$.

  si *u n'est pas coloré* **alors**

    si $ColorAdj_{up} = 1$ **alors**

      pour chaque *ep contenant p* **faire**

        si $ColorAdj_{uj} > 0 \; \forall j \in ep$ **alors**

          $Reste_{ep} \leftarrow Reste_{ep} - Credits_u$.

tant que *OK=VRAI* **faire**

  pour chaque *ensemble ep* **faire**

    si $\sum_{j \in ep} CreditsActuels_j + Reste_{ep} < \sum_{j \in ep} L_j$ **alors**

      OK $\leftarrow$ FAUX.

pour chaque *contrainte* $q \in CtrL_v$ **faire**

  si *couleur p n'est pas qualifiée pour la contrainte* CQ_{qr} **alors**

    $Reserve_{CQ_{qr}} \leftarrow Reserve_{CQ_{qr}} - 1$.

    si $Reserve_{CQ_{qr}} < 0$ **alors**

      OK $\leftarrow$ FAUX.

si *OK=VRAI* **alors**

  si *il y a une contrainte CVMax pour la couleur p* **alors**

    si $CreditsActuels_p > U_p$ **alors**

      OK $\leftarrow$ FAUX.

retourner *OK*

Figure 8.4 – L'algorithme `AffecterCouleur`.

nant la couleur p, nous testons si u a au moins un voisin ayant la couleur j pour chaque $j \in ep$ (i.e., si $ColorAdj_{uj} > 0 \; \forall j \in ep$) ; si tel est le cas nous soustrayons le nombre de crédits de vols de u, $Credits_u$, à la variable $Reste_{ep}$ car u ne peut recevoir aucune couleur j appartenant à ep sans créer de conflit.

- Finalement, nous testons si la somme des crédits actuels des couleurs appartenant à ep plus $Reste_{ep}$ est strictement plus petite que la somme des nombres minimaux de crédits de vols des couleurs de ep (i.e., si $\sum_{j \in ep} CreditsActuels_j + Reste_{ep} < \sum_{j \in ep} L_j$). Si tel est le cas, alors nous effectuons un retour-arrière car cela signifie que pour les couleurs de ep il n'est pas possible d'attribuer une couleur aux sommets non encore colorés de telle sorte que les contraintes de crédits de vol minimum pour les couleurs de ep soient toutes satisfaites.

La procédure consistant à enlever la couleur p de v est présentée dans la Figure 8.5.

La procédure de désaffectation fait le processus inverse de la procédure d'affectation. Quand nous désaffectons la couleur p du sommet v nous devons diminuer le nombre de crédits actuels effectués par la couleur p, $CreditsActuels_p$, en lui soustrayant le nombre de crédits de v, $Credits_v$. Ensuite, pour tous les ensembles ep qui contiennent au moins une couleur $j \in ep$ telle que v n'a aucun voisin de couleur j (i.e., $ColorAdj_{vj} = 0$), nous mettons à jour $Reste_{ep}$ en lui additionnant $Credits_v$, car v peut prendre cette couleur j sans créer de conflit, donc le nombre de crédits de v peut contribuer à $Reste_{ep}$. Tous les voisins u de v ont un voisin en moins de couleur p, donc nous mettons à jour $ColorAdj_{up}$ en lui soustrayant 1. De plus, si un voisin u de v n'est pas coloré et s'il n'a maintenant aucun voisin de couleur p (i.e., $ColorAdj_{up} = 0$) alors pour tous les sous-ensembles ep contenant la couleur p, si u a au moins un voisin de couleur j différente de p pour toutes les couleurs j appartenant à ep (i.e., $ColorAdj_{uj} > 0 \; \forall j \neq p$ dans ep) alors u peut maintenant contribuer à atteindre le nombre minimum de crédits pour le sous-ensemble ep en lui attribuant la couleur p (alors qu'il ne le pouvait pas quand v avait la couleur p)

`EnleverCouleur` (v, p)

Entrée : Un sommet v et une couleur p.
enlever p de v.
$CreditsActuels_p \leftarrow CreditsActuels_p - Credits_v$.
pour chaque *ep contenant au moins une couleur j telle que* $ColorAdj_{vj} = 0$ **faire**
  $Reste_{ep} \leftarrow Reste_{ep} + Credits_v$.
pour chaque *sommet u adjacent à v* **faire**
  $ColorAdj_{up} \leftarrow ColorAdj_{up} - 1$.
  si *u n'est pas coloré* **alors**
    si $ColorAdj_{up} = 0$ **alors**
      pour chaque *ep contenant la couleur p* **faire**
        si $ColorAdj_{uj} > 0 \; \forall j \neq p, \; j \in ep$ **alors**
          $Reste_{ep} \leftarrow Reste_{ep} + Credits_u$.
pour chaque *contrainte* $CQ_{qr} \in CtrL_v$ **faire**
  si *couleur p n'est pas qualifiée pour la contrainte* CQ_{qr} **alors**
    $Reserve_p \leftarrow Reserve_p + 1$.

Figure 8.5 – L'algorithme `EnleverCouleur`.

et $Reste_{ep}$ est donc augmenté de $Credits_u$.

L'algorithme de branchement, appelé `Color`, est présenté dans la Figure 8.6.

Cet algorithme est récursif. Il prend en entrée la profondeur actuelle de l'arbre de séparation et évaluation progressive, notée d. Si d est plus grand que le nombre de sommets dans le graphe cela signifie que l'algorithme de branchement a atteint une feuille et donc qu'une solution a été trouvée. À ce moment-là, l'algorithme retourne FAUX et tous les appels récursifs antérieurs retournent FAUX et donc l'algorithme s'arrête. Sinon, à chaque itération, l'algorithme sélectionne le sommet v ayant le plus de couleurs adjacentes et essaie d'affecter une couleur p appartenant à D_v. L'algorithme appelle alors la procédure `AffecterCouleur` (v, p) qui affecte la couleur au sommet et vérifie les contraintes de qualification et de crédits de vol minimum et maximum (comme nous l'avons vu ci-dessus). Si l'affectation ne crée pas de conflit avec une contrainte de quali-

```
Color(d)
Entrée : La profondeur d.
Sortie : VRAI s'il n'existe pas de coloration satisfaisant toutes les contraintes (i.e., si
         l'algorithme a exploré tous les noeuds possible de l'arbre et n'a pas trouvé de
         solution), FAUX s'il a trouvé une coloration satisfaisant toutes les contraintes.
si d ≥ N alors
    retourner FAUX
sinon
    Soit v le sommet ayant le plus de couleurs adjacentes.
    L ← L ∪ {v}.
    Continuer ← VRAI.
    pour j = 1 à |D(V)| faire
        si v peut avoir la couleur j alors
            si AffecterCouleur(v, j) alors
                Continuer ← Color(d + 1).
                si Continuer=FAUX alors
                    retourner FAUX
            EnleverCouleur(v, j).
    L ← L \ {v}
    retourner VRAI
```

Figure 8.6 – L'algorithme `Color`.

fication ou avec une contrainte de crédit de vol minimum ou maximum, alors il continue en essayant d'affecter une couleur à un nouveau sommet en appelant $Color(d+1)$. Sinon, il effectue un retour-arrière et appelle `EnleverCouleur` (v, p) afin d'enlever la couleur p au sommet v et de mettre à jour les variables servant à vérifier les contraintes.

La procédure `Dsatur` modifiée est présentée dans la Figure 8.7.

La procédure `Dsatur modifié` initialise d'abord les différentes valeurs. Puis les couleurs sont renumérotées afin que les couleurs pour lesquelles il y a une contrainte de crédit de vol minimum aient les plus petits numéros et à ce que tous les numéros de couleurs utilisés soient contigus. En effet, si nous ne faisons pas cette renumérotation, l'algorithme de branchement va d'abord essayer de colorer les sommets avec les couleurs

Algorithme `Dsatur modifié`

Entrée : Un problème de D-coloration avec contraintes additionnelles de qualification, et contraintes de crédits de vol minimal et maximal.
Sortie : *VRAI* si le problème est non réalisable, et *FAUX* sinon.
`Initialisation;`
`RenumeroterCouleurs;`
Création de l'ensemble de couleurs EP et des sous-ensembles ep;
pour chaque *couleur p possible* **faire**
  $CreditsActuels_p \leftarrow 0$.
pour chaque *contrainte de qualification* CQ_{qr} **faire**
  $Reserve_{CQ_{qr}} \leftarrow$ nombre de sommets/tâches de la rotation r - N_{qr}.
pour chaque $ep \in EP$ **faire**
  $Reste_{ep} \leftarrow$ somme des durées des tâches qui peuvent obtenir une couleur $j \in ep$.
pour chaque *sommet* v *et chaque couleur* p **faire**
  $ColorAdj_{vp} = 0$.
`TransformGraph;`
Colorier les sommets de la clique :
pour chaque *sommet* v *de la clique ajoutée* **faire**
  Affecter la couleur correspondante à v.
  pour chaque *sommet* w *adjacent à* v **faire**
    $ColorAdj_{w,couleur(v)} \leftarrow ColorAdj_{w,couleur(v)} + 1$.
  $L \leftarrow L \cup \{v\}$.
Appel de l'arbre de séparation et évaluation progressive :
$Resultat \leftarrow$ `Color` $(|D(V)|)$.
retourner $Resultat$.

Figure 8.7 – L'algorithme `Dsatur` modifié

de plus petits numéros, ce qui fait que l'algorithme doit aller loin dans l'exploration de l'arbre avant de détecter une incohérence et d'effectuer un retour-arrière ou de trouver une solution vérifiant les contraintes. En effectuant la renumérotation, l'algorithme va d'abord essayer d'attribuer une couleur correspondant à une contrainte de crédits de vol minimum. Les incohérences pour cette contrainte sont ainsi détectées plus rapidement et les solutions sont aussi détectées plus rapidement. La différence de temps entre le fait de ne pas faire ou de faire la renumérotation peut être très significative dans certains cas. Par exemple, pour un sous-problème obtenu avec l'algorithme `Insertion` qui conte-

nait uniquement 4 contraintes de crédits de vol minimum, avant renumérotation nous n'avions pas de réponse en 4 jours de calcul, alors qu'après la renumérotation, en 0.02s nous savions que le sous-problème était réalisable.
Puis, la procédure crée les sous-ensembles de couleurs tels que nous les avons décrits ci-dessus. Ensuite, la procédure initialise les variables utilisées pour vérifier les contraintes de crédits de vol minimum et maximum et les contraintes de qualification : $CreditsActuels_p$, $Reserve_{CQ_{qr}}$, $Reste_{ep}$, $ColorAdj_{vp}$. Ensuite, la procédure crée le nouveau graphe en ajoutant la clique comme présenté précédemment (`Transform-Graph`). Puis elle colore les sommets de la clique. Et finalement elle appelle l'algorithme de branchement `Color` afin d'essayer de colorer les autres sommets.

Dans une version précédente de cet algorithme modifié, nous faisions les vérifications des contraintes de crédits de vol minimum seulement quand une feuille était atteinte. Mais, dans certains cas, cela demandait beaucoup trop d'itérations et beaucoup trop de temps pour tester si le problème entré était réalisable ou non. Quand nous avons ajouté les vérifications comme nous les avons présentées ci-dessus, le gain a été significatif en termes de temps de calcul et de nombre d'itérations. Nous allons appeler cette première version `Dsatur modifié 2` et nous allons présenter quelques résultats concernant son utilisation dans la section 8.4.1.3. Ensuite, cette version ne sera plus testée.
Néanmoins, même avec toutes ces astuces pour accélérer la détection rapide des incohérences, il peut arriver que pour certains des IIS-C testés le temps de vérification soit très long. En effet, comme la structure de l'algorithme `Dsatur modifié` est un arbre de séparation et d'évaluation progressive, cet algorithme peut avoir des temps d'exécution très grands selon la taille de l'instance et selon le nombre de couleurs possibles par variable. Notamment, nous ne pourrions pas l'utiliser directement à l'intérieur des algorithmes `HittingSet` ou `Insertion` avec les instances que nous avons testées.

8.4 Résultats expérimentaux

Dans cette section nous présentons les différentes expériences que nous avons effectuées pour tester nos algorithmes de recherche de sous-ensembles incohérents minimaux pour le problème de confection d'horaires pour le personnel navigant aérien. Comme nous l'avons déjà dit, nous avons testé seulement deux des trois méthodes de détection d'IIS et nous cherchons uniquement des IIS de contraintes.

1. La première méthode utilise d'abord l'algorithme `PreFiltering` pour réduire le nombre de contraintes, puis utilise l'algorithme `Insertion`. Elle est nommée "`P+Insertion`". Les algorithmes `PreFiltering` et `Insertion` utilisent l'algorithme tabou décrit à la section 8.2 comme procédure `HMIN` ainsi que l'heuristique basée sur le poids du voisinage (présentée à la section 4.2.2) et la technique d'accélération de la recherche décrite à la section 4.2.3.
2. La deuxième méthode utilise l'algorithme `HittingSet` pour essayer de trouver des IIS de cardinalité minimum. L'algorithme `HittingSet` utilise l'algorithme tabou comme procédure `HMIN` et résout le programme linéaire en nombre entiers décrit à la section 4.4 au moyen de CPLEX 8.1 pour résoudre le problème de hitting set minimum. Nous utilisons la notation "`HS-C`" pour faire référence à cette méthode.

Nous présentons différents résultats obtenus sur des instances modifiées ou créées aléatoirement. Nous ne disposons malheureusement pas d'instance provenant d'un problème réel. Les instances sont séparées en deux groupes. Le premier groupe a principalement servi à tester nos méthodes et à vérifier si nous pouvions gagner du temps lors de la recherche d'IIS en utilisant d'abord le filtrage pour obtenir la cohérence de domaine pour les contraintes de non-chevauchement (filtrage de la contrainte SomeDifferent que nous avons présenté dans le chapitre 6). Ces instances-là ne sont pas proches d'un problème réel et, pour certains tests, nous n'avons pas inclus tous les types de contraintes. Le deuxième groupe d'instances a été créé pour avoir des instances plus proches de la

réalité. Toutes les expériences ont été effectuées sur un PC Pentium D 2.80GHz avec 1024K de cache fonctionnant avec Linux Centos 2.6.9. Pour chaque instance et chaque méthode, cinq relances ont été effectuées pour les instances du premier groupe et dix relances ont été effectuées pour les instances du deuxième groupe. Les résultats donnés dans les tableaux constituent les moyennes des relances effectuées. Si la variance entre le pire et le meilleur résultat est significative, ces deux valeurs seront précisées dans le texte. De même, si nous avons obtenu des échecs parmi les relances, ceci sera aussi précisé dans le texte. Les différents résultats obtenus sont détaillés dans les sections ci-dessous.

Dans tous les tableaux nous allons employer les notations suivantes :

- **TM** : nombre de tâches-mères (= rotations),
- **m** : nombre de couleurs (=employés),
- **TF** : nombre de tâches-filles (= nombre de sommets),
- **Ar** : nombre d'arêtes (= nombre de contraintes de non chevauchement) de l'instance de départ ou de l'IIS,
- **CQ** : nombre de contraintes de qualification de l'instance originale ou de l'IIS,
- **CMin** : nombre de contraintes de crédits de vol minimum faisant partie de l'IIS,
- **CMax** : nombre de contraintes de crédits de vol maximum faisant partie de l'IIS,
- **Tot** : nombre total de contraintes de l'IIS,
- **V** : nombre de sommets (=tâches) impliqués dans les contraintes de l'IIS,
- **Temps Rech** : temps total pour trouver l'IIS (en secondes) (incluant le temps nécessaire à Cplex dans le cas de `HS-C`),
- **Temps Ver** : temps en secondes pour vérifier l'IIS avec l'algorithme exact `Dsatur modifié`,
- **Temps Cplex** : temps uniquement de CPLEX (en secondes) utilisé par `HS-C`,
- **HS-C** : algorithme `HittingSet` en mode contraintes,
- **P+Insertion** : algorithme `Prefiltering+Insertion`,
- **Nb p** : nombre de couples sommet-couleur filtrés par l'algorithme filtrant les do-

maines des variables par rapport aux contraintes de non chevauchement,

- **Moy Temps** : temps du filtrage des domaines pour les contraintes de non chevauchement (correspondant à la contrainte SomeDifferent).

8.4.1 Premier groupe d'instances

Dans cette section, nous présentons des résultats sur des instances qui ne sont pas toutes similaires à un problème de confection d'horaires pour le personnel navigant aérien. Néanmoins, ces résultats nous ont permis de tester nos algorithmes sur plusieurs types d'instances et nous ont aussi permis, pour les deux premiers jeux de données, de tester si le temps de recherche d'un IIS était plus petit ou plus grand que le temps cumulé du filtrage pour les domaines des contraintes de non-chevauchement et de la recherche d'un IIS sur le problème filtré. Nous allons aussi présenter dans la section 8.4.1.3 quelques tests avec la version numéro 2 de `Dsatur modifié` (i.e., dans laquelle la vérification des contraintes de crédits de vol minimum se fait seulement dans les feuilles de l'arbre) afin de justifier pourquoi nous avons amélioré cette méthode.
Ce premier groupe d'instances est subdivisé en trois sous-groupes. Les instances du premier sous-groupe (présentées dans la section 8.4.1.1) ont été obtenues à partir des graphes ayant une unique D-coloration que nous avons présentés dans la section 6.2.3 auxquels nous avons ajouté des contraintes de qualification uniquement (il n'y a pas de contraintes de crédits de vol minimum ou maximum). Les instances du deuxième sous-groupe (présentées dans la section 8.4.1.2) sont des petites instances générées aléatoirement ayant aussi uniquement des contraintes de qualification et des contraintes de non chevauchement. Les instances du troisième sous-groupe (présentées dans la section 8.4.1.3) sont des instances générées aléatoirement ayant tous les types de contraintes mais pour lesquelles le nombre de crédits de vol d'une tâche est égal à sa durée et cette valeur n'est pas proche des valeurs rencontrées en réalité pour un problème habituel de confection d'horaires pour le personnel navigant aérien, mais qui pourrait être proche

pour un autre problème de confection d'horaires. Pour tous les tests réalisés sur ce premier groupe d'instances, la fonction objectif f_ω a été utilisée comme fonction objectif de l'algorithme de recherche tabou. Bien entendu, pour les instances ne comportant pas de contraintes de crédits de vol minimum et maximum, cette fonction objectif f_ω comporte uniquement les termes concernant les contraintes de non-chevauchement f_a et les contraintes de qualification $f_{CQ_{qr}}$.

8.4.1.1 Instances ayant une unique D-coloration

Pour les premières expériences, nous avons repris les instances ayant une unique D-coloration présentées à la section 6.2.3 et nous avons ajouté pour chacune d'entre elles une contrainte de qualification. Afin de nous assurer de la non-réalisabilité des instances créées, nous avons utilisé la connaissance que nous avons de ces instances. Les contraintes de qualification ajoutées demandent toujours qu'un sommet v ait une couleur d'une certaine qualification et nous nous assurons que la seule valeur de D_v participant à l'unique solution n'ait pas cette qualification. Comme exemple, nous reprenons le graphe uniquely pour $k = 2$ présenté dans la figure 8.8.

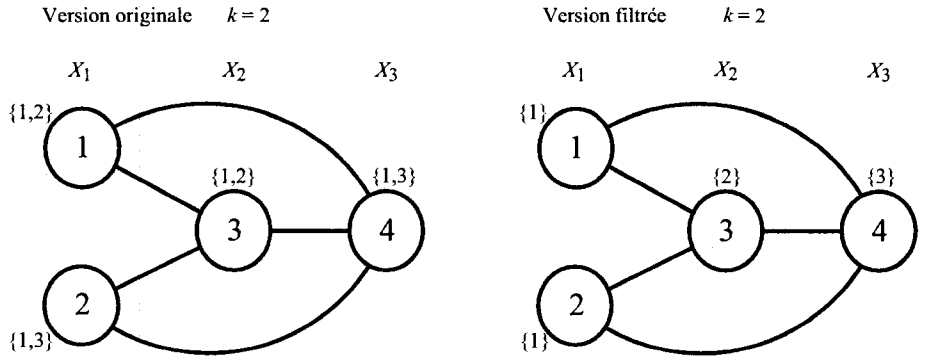


Figure 8.8 – Graphe ayant une unique D-coloration pour $k = 2$. Le graphe de gauche est la version originale du graphe et le graphe de droite est la version filtrée.

Une contrainte de qualification possible est de demander que le sommet 2 ait une couleur ayant la qualification q et seules les couleurs 2 et 3 aient la qualification q. Donc

ce problème est irréalisable. En effet, si nous examinons uniquement les contraintes de non chevauchement et que nous filtrons les domaines des sommets pour cette contrainte, la seule solution possible pour le sommet 2 est la couleur 1 et cette couleur n'a pas la qualification demandée.

Pour certaines de ces instances, nous avons créé différentes versions avec des contraintes de qualification qui ne concernent pas les mêmes sommets. Ces expériences nous ont permis de tester les deux algorithmes `Insertion` et `HittingSet` sur des petites instances et nous ont permis de vérifier si le filtrage des domaines concernant les contraintes de non-chevauchement pouvait faire gagner du temps pour la recherche d'IIS. Les résultats sont présentés dans le tableau 8.1.

Nous donnons des résultats pour le temps de filtrage des domaines des contraintes de non-chevauchement ainsi que pour les tailles des IIS-C obtenus et les temps de recherche d'un IIS dans les quatre cas suivants : `HS-C` sans filtrer les domaines, `HS-C` après avoir filtré les domaines, `P+Insertion` sans filtrer les domaines et `P+Insertion` après avoir filtré les domaines.

Le temps limite de recherche d'IIS avec l'algorithme `HS-C` a été fixé à 3600 secondes (1 heure). Pour une seule des instances : uniquely_k9_v4, `HS-C` n'a pas résolu l'instance en une heure. Dans ce cas-là, le résultat donné dans la colonne "tot" est une borne inférieure sur le nombre de contraintes de l'IIS-C. Tous les IIS-C ont été vérifiés avec l'algorithme `Dsatur modifié`. Les temps de vérification des IIS-C trouvés sans filtrer les domaines sont très petits : entre 0s et 1.2s selon l'instance. Les temps de vérification des IIS-C trouvés après avoir filtré les domaines ne sont pas précisés dans le tableau car ils valent tous 0s (ce qui est très logique car l'IIS-C après avoir filtré les domaines contient toujours une seule contrainte (de qualification)).

L'instance "uniquely_k2" correspond à l'exemple de la Figure 8.8. Nous pouvons remarquer que l'IIS-C de cardinalité minimum de l'instance non filtrée ne contient pas toutes

Tableau 8.1 – Résultats pour les instances ayant une unique D-coloration modifiées par l'ajout de contraintes de qualification.

Instance					Filtrage	HS-C avant Filtrage						HS-C après Filtrage						P+Insertion avant Filtrage						P+Insertion après Filtrage					
					Temps	IIS-C				Temps	Temps	IIS-C				Temps		IIS-C				Temps	Temps	IIS-C				Temps	
Nom	Var	m	Ar	CQ	t	tot	Ar	CQ	V	Rech	Ver	tot	Ar	CQ	V	Rech	tot	Ar	CQ	V	Rech	Ver	tot	Ar	CQ	V	Rech	
uniquely_k2	5	2	5	1	0	5	4	1	4	0.14	0	1	0	1	1	0.03	5	4	1	4	0.13	0	1	0	1	1	0.03	
uniquely_k3_v1	7	3	19	1	0.03	8	7	1	5	0.19	0	1	0	1	1	0.04	8	7	1	5	0.31	0	1	0	1	1	0.04	
uniquely_k3_v2	7	3	19	1	0.03	12	11	1	7	0.25	0	1	0	1	1	0.04	12	11	1	7	0.46	0	1	0	1	1	0.04	
uniquely_k3_v3	7	3	19	1	0.03	10	9	1	5	0.29	0	1	0	1	1	0.05	10	9	1	5	0.51	0	1	0	1	1	0.05	
uniquely_k4_v1	10	4	42	1	0.11	19	18	1	7	0.67	0	1	0	1	1	0.06	19	18	1	7	0.61	0	1	0	1	1	0.06	
uniquely_k5_v1	13	5	74	1	0.14	16	15	1	6	0.51	0	1	0	1	1	0.04	16	15	1	6	0.65	0	1	0	1	1	0.05	
uniquely_k5_v2	13	5	74	1	0.14	17	16	1	7	0.7	0	1	0	1	1	0.05	17	16	1	7	0.69	0	1	0	1	1	0.05	
uniquely_k5_v3	13	5	74	1	0.14	33	32	1	9	2.06	0	1	0	1	1	0.05	43	42	1	11.25	2.04	0	1	0	1	1	0.05	
uniquely_k5_v4	13	5	74	1	0.14	44	43	1	12	1.15	0	1	0	1	1	0.05	44	43	1	12	1.58	0	1	0	1	1	0.05	
uniquely_k5_v5	13	5	74	1	0.14	16	15	1	6	0.58	0	1	0	1	1	0.05	16	15	1	6	0.66	0	1	0	1	1	0.04	
uniquely_k6_v1	16	6	115	1	0.29	23	22	1	8	1.3	0	1	0	1	1	0.05	23	22	1	8	1	0	1	0	1	1	0.05	
uniquely_k6_v2	16	6	115	1	0.29	33	32	1	9	1.96	0	1	0	1	1	0.05	36.6	35.6	1	9.6	1.95	0	1	0	1	1	0.05	
uniquely_k6_v3	16	6	115	1	0.29	22	21	1	7	1.17	0	1	0	1	1	0.05	22	21	1	7	0.96	0	1	0	1	1	0.05	
uniquely_k6_v4	16	6	115	1	0.29	23	22	1	8	1.16	0	1	0	1	1	0.05	23	22	1	8	0.9	0	1	0	1	1	0.05	
uniquely_k7_v1	19	7	165	1	0.66	1	0	1	1	0.17	0	1	0	1	1	0.05	1	0	1	1	0.22	0	1	0	1	1	0.05	
uniquely_k7_v2	19	7	165	1	0.66	29	28	1	8	1.8	0.01	1	0	1	1	0.06	29	28	1	8	1.35	0.01	1	0	1	1	0.05	
uniquely_k7_v3	19	7	165	1	0.66	30	29	1	9	2.39	0	1	0	1	1	0.06	30	29	1	9	1.3	0.01	1	0	1	1	0.05	
uniquely_k7_v4	19	7	165	1	0.66	30	29	1	9	2.2	0.01	1	0	1	1	0.05	30	29	1	9	1.38	0.01	1	0	1	1	0.05	
uniquely_k7_v5	19	7	165	1	0.66	29	28	1	8	1.96	0.01	1	0	1	1	0.06	29	28	1	8	1.38	0.01	1	0	1	1	0.05	
uniquely_k8_v1	22	8	224	1	4.82	37	36	1	9	4.73	0.08	1	0	1	1	0.06	37	36	1	9	1.91	0.08	1	0	1	1	0.06	
uniquely_k8_v2	22	8	224	1	4.82	38	37	1	10	4.63	0.07	1	0	1	1	0.06	38	37	1	10	1.92	0.07	1	0	1	1	0.06	
uniquely_k8_v3	22	8	224	1	4.82	81	80	1	14	197.4	0.12	1	0	1	1	0.06	81	80	1	14	4.26	0.12	1	0	1	1	0.06	
uniquely_k8_v4	22	8	224	1	4.82	93	92	1	17	637.82	0.6	1	0	1	1	0.06	c	-	-	-	2.1	-	1	0	1	1	0.06	
uniquely_k8_v5	22	8	224	1	4.82	63	62	1	12	48.39	0.09	1	0	1	1	0.06	63	62	1	12	3.56	0.09	1	0	1	1	0.06	
uniquely_k9_v1	25	9	292	1	113.3	46	45	1	10	22.79	0.76	1	0	1	1	0.07	46	45	1	10	2.68	0.78	1	0	1	1	0.06	
uniquely_k9_v2	25	9	292	1	113.3	47	46	1	11	5.21	0.73	1	0	1	1	0.06	47	46	1	11	2.49	0.73	1	0	1	1	0.06	
uniquely_k9_v3	25	9	292	1	113.3	105	104	1	16	3461.93	1.2	1	0	1	1	0.07	c	-	-	-	2.87	-	1	0	1	1	0.06	
uniquely_k9_v4	25	9	292	1	113.3	$\geq$ 115				>1h	-	1	0	1	1	0.06	c	-	-	-	2.65	-	1	0	1	1	0.06	
uniquely_k9_v5	25	9	292	1	113.3	85	84	1	14	1422.18	0.97	1	0	1	1	0.06	c	-	-	-	2.59	-	1	0	1	1	0.06	

les arêtes du graphe original, mais seulement quatre arêtes.

Pour quatre instances, l'algorithme `P+Insertion` n'a pas réussi à trouver l'IIS-C (avant de filtrer les domaines), dans ces cas-là un "e" se trouve dans la colonne donnant le nombre total de contraintes de l'IIS.

Nous pouvons remarquer que pour les instances de ce premier sous-groupe, les temps de recherche des IIS, ainsi que les tailles des IIS avant ou après filtrage sont très différents. Après filtrage des domaines, l'IIS-C contient toujours une seule contrainte de qualification, mais avant le filtrage il contient la contrainte de qualification et plusieurs arêtes. Avant filtrage des domaines, les temps de recherche d'IIS varient entre 0.14s et plus d'une heure pour l'algorithme `HS-C` et entre 0.13s et 4.29s pour l'algorithme `P+Insertion`. Par contre, après le filtrage des domaines, les temps sont beaucoup plus stables : entre 0.03s et 0.07s pour les deux algorithmes. Ceci s'explique certainement par le fait que l'IIS-C de la version filtrée contient toujours uniquement une seule contrainte de qualification touchant un seul sommet. Pour plusieurs instances, si nous additionnons le temps de filtrage des domaines des variables (pour les contraintes SomeDifferent représentant ici le non-chevauchement) avec le temps de recherche d'IIS après filtrage, ce total est plus petit que le temps de recherche d'IIS sur l'instance originale. Ceci démontre que le fait d'effectuer d'abord le filtrage des domaines des variables pour les contraintes SomeDifferent peut parfois nous faire économiser du temps lors de la recherche d'IIS.

8.4.1.2 Petites instances aléatoires avec uniquement des contraintes de qualification et des contraintes de non chevauchement

Après avoir testé les instances ayant une unique D-coloration auxquelles nous avons ajouté des contraintes de qualification, nous avons voulu tester nos méthodes avec des petites instances aléatoires simulant des problèmes de confection d'horaire pour lesquelles il n'y a que des contraintes de non chevauchement et de qualification. Avec ces

instances, nous voulons tester l'efficacité de nos méthodes. De plus, nous voulons vérifier si l'utilisation du filtrage pour les contraintes de non chevauchement apporte un gain lors de la recherche d'IIS comme pour les instances du premier sous-groupe.
Les instances présentées dans les tableaux 8.2 et 8.3 ont été générées aléatoirement et ont uniquement des contraintes de non chevauchement et des contraintes de qualification. Chaque instance a 100 tâches-mères. Les tâches-mères peuvent avoir au plus 5 ou 7 tâches-filles selon les instances. Il y a 8 qualifications possibles et chaque rotation peut avoir au plus 4 contraintes de qualification. Chaque couleur est attribuée aléatoirement au domaine de 20 tâches-mères afin de simuler les tâches préférées que voudrait faire un employé dans le mois. Les durées des tâches sont en jours et varient de 1 à 5 jours. Les crédits de vol sont égaux aux durées. Ces instances sont réalisables si nous considérons uniquement les contraintes de non-chevauchement. Ceci a été vérifié avec un algorithme tabou cherchant une coloration des sommets en considérant uniquement les contraintes de non chevauchement.

Dans le tableau 8.2 sont présentés les résultats concernant la recherche d'IIS avec l'algorithme `P+Insertion` : avant le filtrage des domaines des variables (pour les contraintes SomeDifferent), après le filtrage des domaines des variables, mais en conservant toutes les arêtes de l'instance originale et après le filtrage des domaines et en enlevant les arêtes superflues (ce qui est fait lors de la procédure de `Réduction` initiale effectuée par notre algorithme de filtrage pour les contraintes SomeDifferent qui a été présenté dans la section 6.1.2). Dans le tableau 8.3 sont présentés les mêmes résultats pour l'algorithme `HS-C`. Tous les IIS-C ont été vérifiés avec l'algorithme `Dsatur modifié`. Les temps de vérification n'ont pas été précisés dans les tableaux. En fait, pour les IIS-C trouvés par l'algorithme `P+Insertion` les temps de vérification sont tous compris entre 0s et 0.01s (dans les trois cas testés : sans filtrage, avec filtrage et avec filtrage et propagation) sauf pour l'instance numéro 6 pour laquelle les moyennes

Tableau 8.2 – Résultats pour l'algorithme `P+Insertion` sur les petites instances aléatoires du premier groupe pour lesquelles il n'y a pas de contraintes de crédits de vol minimum et maximum.

							Filtrage		P+Insertion avant filtrage					P+Insertion après filtrage					P+Insertion après filtrage et propa.				
Instance Départ							Moy	Moy	IIS-C				Temps	IIS-C				Temps	IIS-c				Temps
Nom	TM	m	TF	Nb Ar	CQ		Nb p	Temps	Tot	Ar	CQ	V	Rech	Tot	Ar	CQ	V	Rech	Tot	Ar	CQ	V	Rech
I_TM100_E200_Q5_1	100	200	402	12577	153		26	7.88	8.0	6.0	2.0	4.0	1.41	8.0	6.0	2.0	4.0	1.47	8.0	6.0	2.0	4.0	1.32
I_TM100_E200_Q5_2	100	200	368	11036	151		2	6.01	Sat	-	-	-	0.16	Sat	-	-	-	0.13	Sat	-	-	-	0.12
I_TM100_E200_Q5_3	100	200	380	12606	146		89	45.85	4.0	0.0	4.0	3.0	0.46	4.0	0.0	4.0	3.0	0.46	4.0	0.0	4.0	3.0	0.42
I_TM100_E200_Q5_4	100	200	382	12431	146		29	14.21	12.0	10.0	2.0	5.0	1.78	12.0	10.0	2.0	5.0	1.78	12.0	10.0	2.0	5.0	1.52
I_TM100_E200_Q5_5	100	200	371	9990	204		7	0.98	5.0	3.0	2.0	3.0	0.56	5.0	3.0	2.0	3.0	0.57	5.0	3.0	2.0	3.0	0.52
I_TM100_E200_Q5_6	100	200	402	13491	208		100	50.11	2.0	0.0	2.0	6.0	0.74	2.8	0.0	2.8	11.6	0.80	2.8	0.0	2.8	11.6	0.75
I_TM100_E200_Q5_7	100	200	398	14385	199		19	4.38	4.0	0.0	4.0	10.0	0.83	4.0	0.0	4.0	10.0	0.90	4.0	0.0	4.0	10.0	0.83
I_TM100_E200_Q5_8	100	200	407	12241	195		18	2.75	12.0	10.0	2.0	5.0	3.30	12.0	10.0	2.0	5.0	3.17	12.0	10.0	2.0	5.0	2.97
I_TM100_E200_Q5_9	100	200	406	13566	205		31	1.38	8.2	6.0	2.2	5.4	2.62	8.0	6.0	2.0	4.0	2.64	8.0	6.0	2.0	4.0	2.57
I_TM100_E200_Q5_10	100	200	413	12296	198		24	5.86	6.0	3.0	3.0	3.0	2.80	6.4	3.0	3.4	5.4	2.76	6.4	3.0	3.4	5.4	2.66
I_TM100_E200_Q5_11	100	200	393	11822	208		50	22.55	5.4	3.0	2.4	5.4	1.20	5.0	3.0	2.0	3.0	1.19	5.0	3.0	2.0	3.0	1.12
I_TM100_E200_Q7_12	100	200	401	13173	187		31	20.79	2.0	0.0	2.0	5.0	0.84	2.2	0.0	2.2	6.4	0.85	2.2	0.0	2.2	6.4	0.77
I_TM100_E200_Q7_13	100	200	409	14091	195		43	40.65	14.8	0.0	14.8	17.2	1.28	15.4	0.0	15.4	21.2	1.24	15.4	0.0	15.4	21.2	1.20
I_TM100_E200_Q7_14	100	200	375	11861	190		101	50.63	35.6	0.0	35.6	45.8	1.78	35.6	0.0	35.6	45.8	1.77	35.6	0.0	35.6	45.8	1.70
I_TM100_E200_Q7_15	100	200	373	12468	206		41	24.75	7.4	0.6	6.8	7.8	1.38	12.8	1.2	11.6	12.4	1.34	12.8	0.0	12.8	12.4	1.28
I_TM100_E200_Q7_16	100	200	370	11375	151		99	23.78	8.0	0.0	8.0	4.0	0.40	8.0	0.0	8.0	4.0	0.43	8.0	0.0	8.0	4.0	0.37
I_TM100_E200_Q7_17	100	200	414	14085	147		62	51.47	2.0	0.0	2.0	1.0	0.93	2.0	0.0	2.0	1.0	0.92	2.0	0.0	2.2	1.0	0.85
I_TM100_E200_Q5_18	100	200	356	9905	155		92	18.36	8.0	0.0	8.0	8.0	0.61	6.8	0.0	6.8	7.0	0.55	6.8	0.0	6.8	7.0	0.50
I_TM100_E200_Q7_19	100	200	407	13402	139		65	21.01	8.4	0.0	8.4	5.8	0.37	10.0	0.0	10.0	7.0	0.40	10.0	0.0	10.0	7.0	0.34
I_TM100_E200_Q7_20	100	200	383	12347	199		57	21.41	2.4	0.0	2.4	4.8	0.97	2.0	0.0	2.0	2.0	0.49	2.0	0.0	2.2	2.0	0.48
I_TM100_E200_Q5_21	100	200	343	10032	138		84	23.36	4.0	0.0	4.0	4.0	0.44	4.0	0.0	4.0	4.0	0.44	4.0	0.0	4.0	4.0	0.39
I_TM100_E200_Q7_22	100	200	444	15190	192		54	38.13	6.6	0.0	6.6	10.4	0.73	2.2	0.0	2.2	4.2	0.52	2.2	0.0	2.2	4.2	0.48
I_TM100_E200_Q7_23	100	200	427	13211	137		82	39.99	14.6	0.0	14.6	21.4	0.66	1.4	0.0	1.4	4.6	0.31	1.4	0.0	1.3	4.6	0.27
I_TM100_E200_Q7_24	100	200	415	15383	189		43	39.75	17.2	0.0	17.2	21.8	1.05	17.2	0.0	17.2	21.8	1.04	17.2	0.0	17.2	21.8	0.98
I_TM100_E200_Q7_25	100	200	374	11589	146		65	29.30	4.0	0.0	4.0	2.0	0.46	4.0	0.0	4.0	2.0	0.46	4.0	0.0	4.0	2.0	0.40
I_TM100_E200_Q5_26	100	200	336	10464	140		81	21.56	6.0	0.0	6.0	5.0	0.41	6.0	0.0	6.0	5.0	0.38	6.0	0.0	6.0	5.0	0.33
I_TM100_E200_Q7_27	100	200	407	14020	148		105	31.67	2.2	0.0	2.2	6.0	0.62	2.0	0.0	2.0	5.0	0.57	2.0	0.0	2.0	5.0	0.47
I_TM100_E200_Q5_28	100	200	374	11539	151		72	26.89	4.0	0.0	4.0	4.0	0.50	4.0	0.0	4.0	4.0	0.52	4.0	0.0	4.0	4.0	0.45
I_TM100_E200_Q7_29	100	200	415	15468	194		47	34.73	2.6	0.0	2.6	3.6	1.07	2.6	0.0	2.6	3.6	1.02	2.6	0.0	2.6	3.6	0.97
I_TM100_E200_Q7_30	100	200	418	14085	148		80	27.60	2.0	0.0	2.0	4.0	0.48	2.0	0.0	2.0	4.0	0.47	2.0	0.0	2.0	4.0	0.40

Tableau 8.3 – Résultats pour l'algorithme HS-C sur les petites instances aléatoires du premier groupe pour lesquelles il n'y a pas de contraintes de crédits de vol minimum et maximum.

						Filtrage		HS-C avant filtrage					HS-C après filtrage					HS-C après filtrage et propa.				
Instance Départ						Moy	Moy	IIS-C				Temps	IIS-C				Temps	IIS-c				Temps
Nom	TM	m	TF	Ar	CQ	Nb p	Temps	Tot	Ar	CQ	V	Rech	Tot	Ar	CQ	V	Rech	Tot	Ar	CQ	V	Rech
I_TM100_E200_Q5_1	100	200	402	12577	153	26	7.88	8.0	6.0	2.0	4.0	5.70	8.0	6.0	2.0	4.0	6.07	8.0	6.0	2.0	4.0	4.54
I_TM100_E200_Q5_2	100	200	368	11036	151	2	6.01	Sat	-	.	-	0.44	Sat	-	-	-	0.32	Sat	-	-	-	0.24
I_TM100_E200_Q5_3	100	200	380	12606	146	89	45.85	2.0	0.0	2.0	1.8	2.09	2.0	0.0	2.0	1.8	2.08	2.0	0.0	2.0	1.8	1.59
I_TM100_E200_Q5_4	100	200	382	12431	146	29	14.21	12.0	10.0	2.0	5.0	8.99	12.0	10.0	2.0	5.0	9.00	12.0	10.0	2.0	5.0	6.33
I_TM100_E200_Q5_5	100	200	371	9990	204	7	0.98	3.0	1.0	2.0	3.0	2.14	3.0	1.0	2.0	3.0	2.03	3.0	1.0	2.0	3.0	1.68
I_TM100_E200_Q5_6	100	200	402	13491	208	100	50.11	2.0	0.0	2.0	6.0	5.49	2.0	0.0	2.0	6.0	4.64	2.0	0.0	2.0	6.0	3.89
I_TM100_E200_Q5_7	100	200	398	14385	199	19	4.38	2.0	0.0	2.0	5.2	4.14	2.0	0.0	2.0	6.0	7.37	2.0	0.0	2.0	6.0	5.97
I_TM100_E200_Q5_8	100	200	407	12241	195	18	2.75	12.0	10.0	2.0	5.0	23.74	12.0	10.0	2.0	5.0	21.78	12.0	10.0	2.0	5.0	18.10
I_TM100_E200_Q5_9	100	200	406	13566	205	31	1.38	8.0	6.0	2.0	4.0	6.86	8.0	6.0	2.0	4.0	6.73	8.0	6.0	2.0	4.0	6.29
I_TM100_E200_Q5_10	100	200	413	12296	198	24	5.86	6.0	3.0	3.0	3.0	180.43	6.0	3.0	3.0	3.0	192.88	6.0	3.0	3.0	3.0	160.79
I_TM100_E200_Q5_11	100	200	393	11822	208	50	22.55	5.0	3.0	2.0	3.0	9.36	5.0	3.0	2.0	3.0	8.34	5.0	3.0	2.0	3.0	6.92
I_TM100_E200_Q7_12	100	200	401	13173	187	31	20.79	2.0	0.0	2.0	5.0	8.15	2.0	0.0	2.0	5.0	9.42	2.0	0.0	2.0	5.0	7.14
I_TM100_E200_Q7_13	100	200	409	14091	195	43	40.65	2.0	0.0	2.0	1.8	6.38	2.0	0.0	2.0	2.0	6.35	2.0	0.0	2.0	2.0	5.20
I_TM100_E200_Q7_14	100	200	375	11861	190	101	50.63	2.0	0.0	2.0	3.0	8.09	2.0	0.0	2.0	3.0	8.04	2.0	0.0	2.0	3.0	6.17
I_TM100_E200_Q7_15	100	200	373	12468	206	41	24.75	2.0	0.0	2.0	2.0	6.76	2.0	0.0	2.0	1.6	8.77	2.0	0.0	2.0	1.6	6.91
I_TM100_E200_Q7_16	100	200	370	11375	151	99	23.78	2.0	0.0	2.0	1.0	2.14	2.0	0.0	2.0	1.0	1.77	2.0	0.0	2.0	1.0	1.16
I_TM100_E200_Q7_17	100	200	414	14085	147	62	51.47	2.0	0.0	2.0	1.0	7.83	2.0	0.0	2.0	1.0	8.61	2.0	0.0	2.0	1.0	7.14
I_TM100_E200_Q5_18	100	200	356	9905	155	92	18.36	2.0	0.0	2.0	1.8	3.42	2.0	0.0	2.0	1.8	4.14	2.0	0.0	2.0	1.8	2.93
I_TM100_E200_Q7_19	100	200	407	13402	139	65	21.01	2.0	0.0	2.0	1.8	1.94	2.0	0.0	2.0	1.6	1.81	2.0	0.0	2.0	1.6	1.15
I_TM100_E200_Q7_20	100	200	383	12347	199	57	21.41	2.0	0.0	2.0	2.6	5.59	2.0	0.0	2.0	2.0	8.01	2.0	0.0	2.0	2.0	6.27
I_TM100_E200_Q5_21	100	200	343	10032	138	84	23.36	2.0	0.0	2.0	2.0	2.84	2.0	0.0	2.0	2.0	2.85	2.0	0.0	2.0	2.0	1.93
I_TM100_E200_Q7_22	100	200	444	15190	192	54	38.13	2.0	0.0	2.0	4.4	5.05	2.0	0.0	2.0	4.0	6.66	2.0	0.0	2.0	4.0	5.12
I_TM100_E200_Q7_23	100	200	427	13211	137	82	39.99	2.0	0.0	2.0	4.2	2.68	1.0	0.0	1.0	3.0	1.58	1.0	1.0	1.0	3.0	1.09
I_TM100_E200_Q7_24	100	200	415	15383	189	43	39.75	2.0	0.0	2.0	2.2	6.42	2.0	0.0	2.0	2.2	6.42	2.0	0.0	2.0	2.2	4.84
I_TM100_E200_Q7_25	100	200	374	11589	146	65	29.30	2.0	0.0	2.0	1.0	2.38	2.0	0.0	2.0	1.0	2.36	2.0	0.0	2.0	1.0	1.67
I_TM100_E200_Q5_26	100	200	336	10464	140	81	21.56	2.0	0.0	2.0	1.8	1.79	2.0	0.0	2.0	2.2	1.51	2.0	0.0	2.0	2.2	1.00
I_TM100_E200_Q7_27	100	200	407	14020	148	105	31.67	2.0	0.0	2.0	5.0	5.00	2.0	0.0	2.0	5.0	5.10	2.0	0.0	2.0	5.0	3.34
I_TM100_E200_Q5_28	100	200	374	11539	151	72	26.89	2.0	0.0	2.0	2.0	2.95	2.0	0.0	2.0	2.0	2.88	2.0	0.0	2.0	2.0	1.98
I_TM100_E200_Q7_29	100	200	415	15468	194	47	34.73	2.0	0.0	2.0	2.6	12.13	2.0	0.0	2.0	1.6	10.49	2.0	0.0	2.0	1.6	8.24
I_TM100_E200_Q7_30	100	200	418	14085	148	80	27.60	2.0	0.0	2.0	4.0	2.01	2.0	0.0	2.0	4.0	2.12	2.0	0.0	2.0	4.0	1.40

des temps de vérification sont les suivantes : 0.2s pour les IIS-C trouvés sans filtrage des domaines, 0.53s pour les IIS trouvés après filtrage des domaines et 0.54s pour les IIS-trouvés avec filtrage des domaines et propagation des arêtes superflues. Pour les IIS-C trouvés avec l'algorithme `HS-C` les moyennes des temps de vérifications sont comprises entre 0s et 0.03s (dans les trois cas testés) sauf pour l'instance numéro 6 dont le temps de vérification est de 0.2s (dans les trois cas) et pour l'instance numéro 22 dont le temps de vérification est de 0.18s (dans les trois cas). L'instance numéro 2 est réalisable, c'est pour cela que se trouve la mention "Sat" dans la colonne concernant le nombre de contraintes de l'IIS-C. La limite de temps était de 3600 secondes, mais cette limite n'a jamais été atteinte puisque le temps maximal pour l'algorithme `HS-C` est de 192.88s et de 3.3s pour l'algorithme `P+Insertion`.

Nous pouvons remarquer que les IIS-C de ces instances sont très petits, puisque, pour 22 de ces instances, l'IIS-C de cardinalité minimum est de taille 2 (avec 2 contraintes de qualification). L'algorithme `P+Insertion` ne trouve pas toujours l'IIS-C de plus petite taille. Comme nous l'avons précisé ci-dessus, les valeurs données dans le tableau sont les moyennes de cinq relances. Les tailles minimales et maximales des IIS-C ou les valeurs minimales ou maximales des temps ne sont pas précisées dans ce cas-ci, car en général la variance n'est vraiment pas grande. D'ailleurs, pour de nombreuses instances, les cinq relances ont donné pour `P+Insertion` un IIS-C de même taille (rappelons que, comme l'algorithme `HS-C` donne l'IIS-C de cardinalité minimum, il est obligatoire que pour cet algorithme les cinq relances donnent un IIS-C de même taille si l'algorithme termine dans le temps imposé).

Nous pouvons remarquer que, dans ce cas-ci, le filtrage des domaines ne fait pas gagner beaucoup de temps pour la recherche d'un IIS-C et le filtrage en lui-même prend un temps variant de 1.38s à 51.47s. Donc le total du temps de filtrage plus le temps de recherche d'IIS après filtrage n'est pas meilleur que le temps de recherche d'IIS sans filtrage (sauf pour l'instance numéro 10 avec l'algorithme `HS-C`, mais le gain est minime). Le nombre de couples sommet-couleur filtrés varie entre 2 et 105. Pour les

instances des sections ci-dessous, nous n'effectuons plus le filtrage pour les contraintes de non-chevauchement et donc nous ne testerons pas si nous obtenons un gain lors de la recherche d'IIS en filtrant les domaines pour les contraintes de non chevauchement. Les deux raisons pour lesquelles nous n'effectuons plus ce filtrage sont que, premièrement, nous avons pu remarquer avec les instances de cette section que le filtrage n'apportait en général aucun gain de temps, et deuxièmement les instances que nous testons dans les sections suivantes sont de plus grande taille et la transformation consistant à ajouter une clique crée des instances qui dépassent les limites actuelles sur la taille des instances utilisables par `Dsatur` (qui est utilisé par `TestColorability` de la Figure 6.6 pour trouver le nombre chromatique de $G \oplus D$).

8.4.1.3 Instances aléatoires avec tous les types de contraintes

Les instances que nous avons testées dans le sous-groupe précédent ne contiennent pas de contraintes de crédit de vol minimum et maximum et elles sont d'assez petite taille. Afin de tester nos algorithmes avec toutes les contraintes, nous avons donc créé de nouvelles instances de taille un peu plus grande et qui contiennent des contraintes de tous les types.

Les instances dont les résultats sont présentés dans le tableau 8.4 ont été créées aléatoirement de la façon suivante. Il y a 200 employés (donc 200 couleurs) et 300 tâches-mères. Chaque tâche-mère peut avoir au plus 7 tâches-filles (le nombre de tâches-filles obtenues est compris entre 1142 et 1223). Il y a 8 qualifications possibles. Chaque tâche-mère peut avoir au plus 4 contraintes de qualification. Chaque couleur est attribuée aléatoirement au domaine de 20 tâches-mères afin de simuler les tâches préférées que voudrait faire un employé dans le mois. Les durées des tâches sont en jours et varient de 1 à 5 jours. Les crédits de vol sont égaux aux durées. Le nombre de crédits de vol minimum est 15 jours et le nombre de crédits de vol maximum est 20 jours. À nouveau, ces instances sont réalisables si nous considérons uniquement les contraintes de non-chevauchement. Ceci a été

testé avec un algorithme tabou cherchant une D-coloration en considérant uniquement les contraintes de non-chevauchement. Tous les IIS ont été vérifiés avec l'algorithme exact `Dsatur modifié`.

Nous présentons des résultats pour les algorithmes `P+Insertion` et `HS-C` ainsi que les temps de vérification de l'algorithme `Dsatur modifié` et le temps de `Cplex` uniquement lors de son utilisation avec `HS-C`. Le temps limite de recherche d'IIS a été fixé à 36000 secondes (10 heures). Quand l'algorithme `HS-C` a atteint cette limite le temps indiqué dans la colonne "Temps Rech" est noté "$> 10h$" et le résultat donné sur la taille de l'IIS-C est une borne inférieure sur la taille réelle de l'IIS-C de cardinalité minimum. Les bornes inférieures sont précédées du signe "$\geq$". L'instance numéro 9 est réalisable. La recherche d'IIS avec `P+Insertion` a échoué pour les instances numéro 15 et 18. C'est pour cette raison que se trouve un "e" dans la colonne donnant le nombre total de contraintes de l'IIS-C. Nous pouvons remarquer que les IIS-C des instances 1 à 5 sont de très petite taille. En étudiant la structure de ces IIS-C, nous avons remarqué qu'ils sont formés par les contraintes de non-chevauchement et de qualification sur une seule rotation (donc en termes de graphe les contraintes de non-chevauchement forment une clique). Pour toutes les autres instances, les IIS-C obtenus concernent des contraintes provenant d'au moins deux rotations (même pour l'IIS-C de l'instance numéro 12 qui concerne deux rotations). Nous pouvons remarquer que la taille moyenne des IIS-C trouvés par `P+Insertion` varie de 5 à 68.4 selon les instances et celle trouvés par `HS-C` varie de 5 à 27 selon les instances (quand l'algorithme termine en moins de 10 heures). Les temps pour trouver un IIS-C sont typiquement d'une dizaine de minutes et varient de 5.38s à 1726.74s pour l'algorithme `P+Insertion`. Le temps le plus long correspond à l'IIS-C de plus grande taille, par contre le temps le plus court ne correspond pas à l'IIS-C de plus petite taille. Quand l'algorithme `HS-C` trouve un IIS-C en moins de 10 heures, ses temps d'exécution varient de 315.9s à 10579.16s.

Tableau 8.4 – Résultats pour des petites instances aléatoires du premier groupe comportant tous les types de contraintes.

Instance Depart						P+Insertion								HS-C								
						IIS-C						Temps	Temps	IIS-C						Temps	Temps	Temps
Nom	TM	m	TF	Nb Ar	CQ	Tot	Ar	CQ	CMin	CMax	V	Rech	Ver	Tot	Ar	CQ	CMin	CMax	Var	Rech	Cplex	Vér
Inst_1	300	200	1216	116469	611	5.0	3.0	2.0	-	-	3.0	210.52	0.0	5.0	3.0	2.0	-	-	3.0	5930.8	4562.82	0.0
Inst_2	300	200	1158	105330	598	8.0	6.0	2.0	-	-	4.0	170.62	0.0	8.0	6.0	2.0	-	-	4.0	2688.91	1844.91	0.0
Inst_3	300	200	1190	112356	636	13.0	10.0	3.0	-	-	5.0	319.61	0.0	≥ 10			-	-		$> 10h$		
Inst_4	300	200	1198	108633	620	10.0	6.0	4.0	-	-	6.0	165.97	0.0	5.0	3.0	2.0	-	-	3.0	2420.2	1683.55	0.0
Inst_5	300	200	1162	101923	589	5.0	3.0	2.0	-	-	3.0	102.41	0.0	5.0	3.0	2.0	-	-	3.0	315.9	114.98	0.0
Inst_6	300	200	1158	105550	598	31.0	28.0	3.0	-	-	8.0	494.31	0.18	≥ 17			-	-		$> 10h$		
Inst_7	300	200	1146	113731	611	37.0	35.0	2.0	-	-	10.0	596.97	7.5	≥ 16			-	-		$> 10h$		
Inst_8	300	200	1218	119141	630	58.0	53.0	5.0	-	-	13.0	1248.38	261.14	≥ 14			-	-		$> 10h$		
Inst_9	300	200	1198	108969	609	Sat	-	-	-	-	-	4.97	-	Sat	-	-	-	-		51.56	-	-
Inst_10	300	200	1190	107373	637	37.0	35.0	2.0	-	-	9.0	5.38	0.39	≥ 19			-	-		$> 10h$		
Inst_11	300	200	1223	121848	628	45.2	42.6	2.6	-	-	10.0	1536.66	0.26	≥ 10			-	-		$> 10h$		
Inst_12	300	200	1152	111215	623	7.0	6.0	1.0	-	-	4.0	138.68	0	7	6.0	1.0	-	-	4.0	1361.91	831.15	0.0
Inst_13	300	200	1211	109581	593	38.0	36.0	2.0	-	-	9.0	506.87	0.11	≥ 17			-	-		$> 10h$		
Inst_14	300	200	1170	104680	611	14.0	10.0	4.0	-	-	5.0	335.54	0	≥ 12			-	-		$> 10h$		
Inst_15	300	200	1180	113010	591	c	-	-	-	-	-	7399.97	-	≥ 11			-	-		$> 10h$		
Inst_16	300	200	1142	104663	595	56.0	55.0	1.0	-	-	11.0	848.52	3.05	≥ 16			-	-		$> 10h$		
Inst_17	300	200	1180	115200	605	68.4	66.0	2.4	-	-	12.8	1726.74	5922.79	≥ 11			-	-		$> 10h$		
Inst_18	300	200	1147	107841	621	c	-	-	-	-	-	14298.18	-	≥ 18			-	-		$> 10h$		
Inst_19	300	200	1181	105842	640	56.0	55.0	1.0	-	-	11.0	625.9	1.35	≥ 20			-	-		$> 10h$		
Inst_20	300	200	1212	112916	634	14.0	10.0	4.0	-	-	5.0	235.1	0.0	≥ 14			-	-		$> 10h$		
Inst_MAX1	300	202	1213	111891	624	15.0	-	13.0	-	2.0	25.0	149.39	0.14	15.0	-	13.0	-	2.0	25.0	10579.16	8256.72	0.21
Inst_MAX2	300	202	1213	111891	624	15.0	-	13.0	-	2.0	25.0	155.22	0.68	15.0	-	13.0	-	2.0	25.0	10264.22	7982.23	0.94
Inst_MIN1	300	205	1219	113840	615	32.8	25.6	5.4	1.8	-	20.8	625.07	0.4	27.0	22.0	3.0	2.0	-	21.0	4731.61	3281.18	44.71
Inst_MIN2	300	205	1219	113840	615	36.0	28.6	5.6	1.8	-	20.8	643.16	32436	27.0	22.0	3.0	2.0	-	21.0	4373.61	2961.85	79351.5
Inst_MIN3	300	205	1219	113840	610	9.75	2.0	6.25	1.5	-	12.25	168.6	2969.75	6.0	3.0	3.0	-	-	3.0	720.04	335.93	0

Nous pouvons remarquer que nos méthodes permettent d'obtenir des IIS-C de taille et de forme non triviale puisque l'IIS-C de plus grande taille comporte plus de 68 contraintes. Nous avons aussi remarqué qu'il arrive que l'algorithme `P+Insertion` trouve aussi l'IIS-C de cardinalité minimum lors des cinq relances (par exemple pour les instances numéro 1, 2, 5, 12, MAX1 et MAX2). Nous avons remarqué lors de l'exécution de ces tests sur les instances numéro 1 à 20 que les IIS-C obtenus ne contiennent jamais de contraintes de crédits de vol minimum et maximum. Ceci est dû à la structure de ces instances : il y a 200 employés pour une moyenne de 1180 tâches et le nombre de crédits de vol moyen des tâches est de 3 jours ce qui donne en moyenne 3540 crédits à effectuer par mois. De plus, le nombre de crédits minimum est $L_p = 15$ et le nombre de crédits maximum est $U_p = 20$. Si nous multiplions ces valeurs par le nombre d'employés, nous obtenons les bornes 3000 et 4000. Donc le nombre moyen de crédits total appartient bien à ces bornes. Ainsi, afin de tester si nos méthodes sont bien capables de trouver des IIS-C contenant des contraintes de crédits de vol minimum ou maximum, nous avons créé artificiellement des instances de telle sorte que leur IIS contienne ces deux types de contraintes. Ces résultats sont ceux des instances MAX1, MAX2 et MIN1, MIN2, MIN3. Ces instances ont été créées à partir de l'instance numéro 9, qui est réalisable, en rajoutant des tâches et des couleurs et en s'assurant que ce ne soit pas réalisable à cause des contraintes de crédits de vol minimum ou maximum combinées aux contraintes de qualification et de non chevauchement. Les tâches ajoutées pour créer l'instance MIN1 sont présentées dans la Figure 8.9, ce sont les tâches 1198 à 1218. Les couleurs ajoutées sont les couleurs 201 à 205. Les couleurs 201 et 202 sont attribuées aux domaines des tâches 1198 à 1203 et à 14 des autres tâches. Les couleurs 203 et 204 sont attribuées aux domaines des tâches 1204 à 1218 et aux domaines de cinq tâches parmi les tâches 1198 à 1203 et la couleur 205 est attribuée aux domaines des tâches 1198 à 1218 sauf en 1205. Tous les domaines des tâches 1198 à 1218 comportent d'autres couleurs. Il y a une contrainte de qualification par tâche 1198 à 1203 demandant une couleur ayant une qualification q et les seules couleurs qui ont cette qualification sont 201 et 202. En fait

cette construction oblige soit la couleur 201 ou la couleur 202 à faire les tâches 1198, 1200 et 1202 et l'autre couleur à faire les tâches 1199,1201 et 1203. De plus, la couleur 203 ou 204 doit effectuer les tâches 1204 à 1208 et l'autre couleur doit faire les tâches 1209 à 1213. Finalement, la couleur 205 doit faire les tâches 1214 à 1218. Or, la somme du nombre des crédits des trois tâches effectuées par les couleurs 201 et 202 est 14 dans les deux cas et le nombre minimum de crédits demandés et 15. C'est la combinaison des contraintes de non-chevauchement, de qualification et de crédits de vols minimum pour les deux couleurs 201 et 202 qui rend l'instance non-réalisable. La différence entre l'instance MIN1 et MIN2 est que dans l'instance MIN2 les domaines des tâches contiennent plus de couleurs (qui ont été enlevées à d'autre tâches). L'instance MIN3 est semblable à l'instance MIN2 pour les domaines des variables, mais les contraintes de qualification sont différentes.

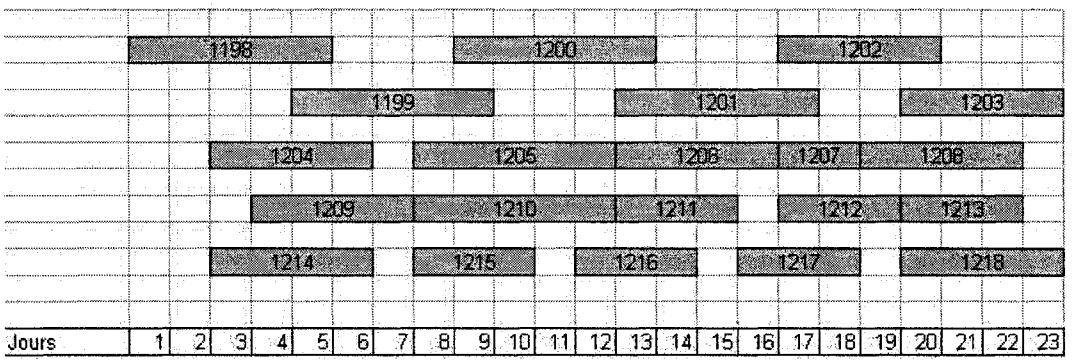


Figure 8.9 – Tâches ajoutées à l'instance numéro 9 afin de créer l'instance MIN1.

Dans la Figure 8.10 sont présentées les tâches qui ont été ajoutées à l'instance numéro 9 afin d'obtenir l'instance MAX1. Les couleurs 201 et 202 sont présentent dans les domaines des tâches 1198 à 1212 et dans les domaines de cinq autres tâches. Il y a une contrainte de qualification par tâche 1198 à 1212 demandant une couleur ayant la qualification q et les seules couleurs ayant cette qualification sont 201 et 202. Donc, la couleur 201 ou 202 doit effectuer les tâches 1198 à 1204 et l'autre couleur doit effectuer

les tâches 1205 à 1212. Or, la somme des crédits de vols de ces tâches est 24 dans le premier cas et 23 dans le deuxième. Ainsi, les contraintes de crédits de vol maximum combinées aux contraintes de non-chevauchement et de qualification pour les tâches 1198 à 1212 sont non réalisables.

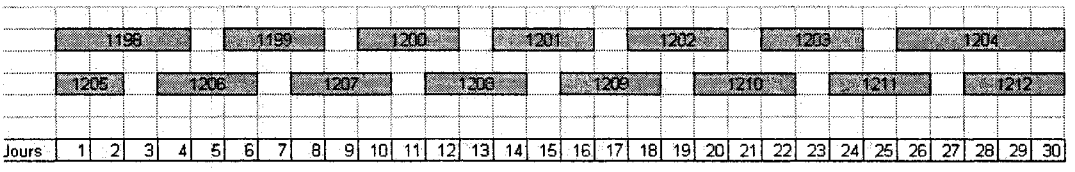


Figure 8.10 – Tâches ajoutées à l'instance numéro 9 afin de créer l'instance MAX1.

Nous remarquons que les IIS-C de ces instances contiennent en moyenne 2 contraintes de crédits de vol maximum pour les instances MAX1 et MAX2 et en moyenne 2 contraintes de crédits de vol minimum pour les instances MIN1 et MIN2 (pour `HS-C`). Nous notons aussi que pour l'instance MIN3, l'algorithme `P+Insertion` trouve des IIS-C contenant en moyenne 1.5 contraintes de crédits de vol minimum, mais que l'IIS-C de cardinalité minimum ne contient pas de contrainte de crédit de vol minimum.

Les temps de vérification des IIS des instances de la Figure 8.4 avec `Dsatur modifié` varient de 0s à 79351.5s, mais pour la plupart des IIS le temps de vérification est très petit.

Afin de justifier l'utilisation de notre version de `Dsatur modifié` par rapport à la version numéro 2 (dans laquelle les contraintes de crédits de vol minimum sont vérifiées dans les feuilles uniquement), nous présentons quelques résultats comparatifs dans le tableau 8.5 pour les instances MIN1 et MIN2 et selon les IIS obtenus en utilisant différentes graines (avec l'algorithme `P+Insertion`).

Nous pouvons remarquer que nous avons un très bon gain de temps en utilisant des vérifications en cours d'exécution pour les contraintes de crédits de vol minimum (par

Tableau 8.5 – Quelques exemples de comparaison des temps de vérification entre la version originale de `Dsatur modifié` et la version 2 de `Dsatur modifié` (i.e., avec vérification des contraintes de crédits de vol minimum dans les feuilles) sur les IIS obtenus en utilisant différentes graines.

Instance	**`Dsatur modifié`** Temps vérif IIS obtenu par `P+Insertion`	**`Dsatur modifié version 2`** Temps vérif IIS obtenu par `P+Insertion`
MIN1 graine 1	0.05	1967.2
MIN1 graine 2	0.02	656.41
MIN1 graine 3	0.7	191741
MIN1 graine 4	0.53	239307
MIN1 graine 5	0.7	191428
MIN2 graine 2	1235.72	21081.58

rapport à la vérification faite dans les feuilles uniquement de la version 2) et en utilisant des vérifications sur des sous-ensembles de couleurs ayant une contrainte de crédits de vol minimum. Nous pouvons aussi remarquer que la relation entre le temps de la version de `Dsatur modifié` que nous avons utilisé et la version 2 n'est pas linéaire. Ceci doit dépendre de la structure des IIS trouvés.

Bien que les instances présentées dans cette section ne modélisent pas précisément la plupart des problèmes de confection d'horaires pour le personnel navigant aérien, elles peuvent très bien modéliser d'autres problèmes de confection d'horaires comme par exemple des problèmes de confection d'horaires pour des industries où les employés travaillent par équipes dans lesquelles il faut qu'un certain nombre d'employés aient certaines qualifications.

8.4.1.4 Conclusion des résultats du premier groupe d'instances

Dans ce premier sous-groupe, nous avons effectué différentes expériences pour tester nos méthodes de recherche d'IIS. Nous avons remarqué que la recherche d'IIS fonctionne bien de façon générale. Comme nous l'avons déjà dit, le nombre d'itérations

de la méthode `HittingSet` peut être exponentiel. C'est pour cette raison que cet algorithme ne finit pas toujours dans le temps limite que nous avons fixé. Néanmoins, cet algorithme nous donne une borne inférieure sur la taille de l'IIS. Généralement, `HittingSet` ne trouve pas un IIS dans la limite de temps, quand l'IIS de cardinalité minimum semble être de grande taille. En effet, en général, c'est lorsque les IIS trouvés par `P+Insertion` sont les plus grands que `HS-C` ne trouve pas un IIS de cardinalité minimum dans le temps imparti.

Nous avons aussi observé que, lorsque les domaines des variables sont filtrés pour les contraintes de non chevauchement (contraintes SomeDifferent), le gain obtenu lors de la recherche d'IIS sur l'instance filtrée dépend du type d'instance. En effet, pour les instances ayant une unique D-coloration, le gain est intéressant pour certaines des instances. Par contre, pour les instances aléatoires que nous avons créées, il n'y a pas de gain de temps.

8.4.2 Deuxième groupe d'instances

Les tâches des instances présentées dans la section 8.4.1.3 ont un nombre de crédits de vol égal à la durée et cette durée est en jours. Or, dans la réalité, les durées sont généralement comptées en minutes et ne sont pas égales au nombre de crédits de vol. En effet, une rotation peut durer 4320 minutes (3 jours) mais le nombre de crédits pour l'employé est de 960 minutes par exemple (2 vols de 8 heures). Ainsi, il faut tenir compte de la durée pour créer les contraintes de non chevauchement, mais il faut tenir compte du nombre de crédits de vol pour les contraintes de crédits de vol minimum et maximum. La relation entre la durée et le nombre de crédits de vol n'est pas linéaire. Comme nous n'avons pas eu d'instances provenant de problèmes réels, nous en avons engendrées qui se rapprochent d'instances réelles au meilleur de notre connaissance. Les instances présentées dans le tableau 8.6 ont 11 qualifications possibles. Le nombre de tâches-mères vaut 300 ou 400 et est indiqué dans le tableau. Chaque tâche-mère a au plus 7 tâches-

filles. Le nombre de couleurs (=employés) est déterminé de telle sorte que le nombre total de crédits de vol de toutes les tâches-filles divisé par le nombre de couleur soit égal à 4500 ($4500 = \frac{L_p+U_p}{2}$ est le nombre moyen de crédits de vol voulu par employé). Pour les instances "B" il y a au plus 4 contraintes de qualification par tâche-mère et pour les instances "C" il y a au plus 5 contraintes de qualification par tâche-mère. Les tâches ont une durée comprise entre 360 et 8950 minutes et le nombre de crédits est compris entre 135 et 2335. Le nombre de crédits de vol minimum L_p vaut 3900 et le nombre de crédits maximum U_p vaut 5100. Chaque couleur est attribuée aléatoirement au domaine de 20 tâches-mères afin de simuler les tâches préférées que voudrait faire un employé dans le mois. Si nous considérons uniquement les contraintes de non-chevauchement, alors les instances sont réalisables. Ceci a été vérifié avec un algorithme tabou cherchant une coloration des sommets en considérant uniquement les contraintes de non-chevauchement.

Tableau 8.6 – Résultats des instances du deuxième groupe.

						P+Insertion							
Instance Depart						IIS-C						Temps	Temps
Nom	TM	m	TF	Nb Ar	CQ	Tot	Ar	CQ	CMin	CMax	V	Rech	Ver
Inst_300_1_C	300	224	1227	162449	697	17.0	15.0	2.0	-	-	6.0	4105.59	0.0
Inst_300_2_C	300	240	1180	160840	698	c	-	-	-	-	-	3480.00	-
Inst_300_3_C	300	264	1270	184634	723	8.1	6.9	1.2	-	-	7.0	1878.97	0.34
Inst_300_4_C	300	236	1134	155946	658	22.0	21.0	1.0	-	-	7.0	4859.87	0.01
Inst_300_5_C	300	266	1252	180996	695	42.0	36.0	6.0	-	-	9.0	10577.40	0.47
Inst_300_6_C	300	249	1215	177441	699	30.0	25.0	5.0	-	-	8.0	5555.19	0.1
Inst_300_7_C	300	238	1170	165034	668	27.86	21.43	5.86	-	0.57	48.86	5131.78	0.01 / >6.5j
Inst_300_8_C	300	227	1165	152397	637	31.0	23.0	7.0	-	1.0	59.0	4285.72	>6.5j
Inst_300_9_C	300	255	1176	167587	683	31.0	28.0	3.0	-	-	8.0	6453.46	0.04
Inst_300_10_C	300	275	1261	186293	734	20.0	16.0	3.0	-	1.0	90.0	4106.91	>6.5j
Inst_300_B	300	240	1244	166717	633	39.0	36.0	3.0	-	-	9.0	4170.28	4.48
Inst_400_B	400	324	1582	291835	839	26.0	22.0	4.0	-	-	8.0	14127.70	0.84

Pour les instances présentées dans la Figure 8.6, nous présentons uniquement les résultats obtenus avec l'algorithme `P+Insertion`, car l'algorithme `HS-C` a permis seulement de trouver des bornes inférieures valant 3 ou 4 sur la taille de l'IIS-C de cardinalité minimum. Donc, ces bornes n'apportent pas une bonne indication de la taille réelle de l'IIS-C de plus petite taille.

Nous pouvons remarquer que pour ces instances, nous avons obtenu un échec pour l'ins-

tance 300_2_C (i.e., les dix relances de l'instance 300_2_C ont échoué). Pour les autres instances, nous avons obtenu des IIS-C qui concernent des contraintes appartenant à au moins deux rotations dans chaque cas. La taille moyenne des IIS-C trouvés varie entre 8.1 et 42.0. La variance entre les différents IIS-C trouvés pour une instance n'est vraiment pas grande puisque pour toutes les instances sauf les instances 300_3_C et 300_7_C toutes les relances réussies ont obtenu un IIS-C de même taille. Pour l'instance 300_3_C la taille de l'IIS-C obtenu varie entre 7 et 15 contraintes. Pour l'instance 300_7_C le nombre de contraintes de l'IIS-C obtenu varie entre 15 et 45 et certains de ces IIS contiennent une contrainte de crédits de vol maximum. Les temps de recherche des IIS varient entre 1878.97s et 14127.70s. Pour les instances 300_1_C et 300_3_C les dix relances ont réussi. Pour les autres instances de ce type, le nombre de réussites varie entre 3 et 8. Les IIS-C des instances 300_1_C, 300_3_C, 300_4_C, 300_5_C et 300_9_C et certains des IIS-C de l'instance 300_7_C ont été vérifiés avec l'algorithme exact en moins de 0.5s. La vérification des instances 300_8_C, 300_10_C et de certains des IIS-C de l'instance 300_7_C a été arrêtée après 6.5 jours. La vérification de ces cas-là a été beaucoup plus longue à cause du fait qu'il y a une contrainte de crédit de vol maximum dans l'IIS-C obtenu. Ainsi, les branches de l'arbre de recherche ne sont coupées à cause de cette contrainte uniquement lorsqu'une affectation partielle viole le nombre maximum de crédits de vol. Donc, la détection des incohérences prend plus de temps. De plus, ces IIS-C comportent beaucoup plus de variables à évaluer que les IIS-C des autres instances de ce type. Nous pouvons donc remarquer que, pour l'instance 300_7_C, la durée de vérification de l'IIS-C obtenu varie grandement selon si l'IIS-C contient ou non une contrainte de crédit de vol maximum. En effet, quand il en contient une le temps de vérification est plus grand que 6.5 jours et quand il n'en contient pas nous avons obtenu des temps de vérification de 0.01s.

8.4.3 Synthèse des résultats

Nous avons présenté des résultats pour deux groupes d'instances. Le premier groupe comprend des instances dont la formulation ne comprend pas forcément tous les types de contraintes et qui ne sont pas proches de problèmes réels. Le deuxième groupe comprend des instances plus proches de problèmes réels et qui comprennent toutes les sortes de contraintes. Nous avons réussi à trouver des IIS-C pour la plupart des instances des deux groupes. Les IIS-C trouvés sont dans certains cas complexes et font souvent intervenir plusieurs tâches-mères (rotations) et deux ou trois sortes de contraintes. Les temps de recherche des IIS-C sont typiquement de quelques secondes pour les instances faciles du premier groupe ne comportant pas toutes les contraintes, d'une dizaine de minutes pour les instances du premier groupe comportant tous les types de contraintes et de plus d'une heure pour les instances du deuxième groupe. De façon générale, les temps sont compris entre 0.14s et 14298s. Ces temps sont plus grands pour les instances du deuxième groupe que pour les instances du premier groupe. Les instances du deuxième groupe sont probablement plus complexes parce que le nombre de qualifications possibles est plus élevé (11 versus 8 pour les instances du premier groupe) et pour les instances "C" le nombre de contraintes de qualification par tâche-mère est plus élevé aussi (5 versus 4). De plus, comme la relation entre le nombre de crédits de vol et la durée du vol n'est ni égale ni linéaire pour les instances du deuxième groupe, ceci peut intervenir dans la relation entre les contraintes de non-chevauchement et de crédits de vol minimum ou maximum. À part les IIS-C de trois instances du deuxième groupe et de l'IIS-C de l'instance MIN2 du premier groupe, les IIS-C des autres instances ont pu être vérifiés dans des temps intéressants avec notre algorithme `Dsatur modifié`. En effet, pour ces autres instances, les temps de vérifications sont compris entre 0s et 5922s et sont généralement de moins d'une seconde. Par contre, pour trois instances du deuxième groupe, après 6.5 jours, l'algorithme `Dsatur modifié` n'avait toujours pas terminé son exécution.

8.5 Conclusion

Dans ce chapitre, nous avons montré comment il est possible d'adapter les algorithmes `Insertion` et `Hitting-Set` pour la recherche d'IIS-C pour le problème de confection d'horaires pour le personnel navigant aérien. Nous avons présenté un algorithme de recherche tabou tenant compte de toutes les contraintes spécifiques du problème de confection d'horaires. Cet algorithme tabou est utilisé par les algorithmes de recherche d'IIS-C. Nous avons aussi détaillé comment nous avons modifié un algorithme exact recherchant le nombre chromatique afin qu'il tienne compte des contraintes de qualification et de crédits de vol maximum et minimum et qu'il indique si l'instance ou l'IIS-C testé est réalisable ou non. Comme nous l'avons précisé, cet algorithme n'est efficace en pratique que sur des petits problèmes. Il sert donc à vérifier les IIS-C obtenus, mais ne peut pas être utilisé à l'intérieur des algorithmes de recherche d'IIS. Ceci constitue donc une grande différence avec le problème SAT pour lequel il existe plusieurs algorithmes exacts très efficaces.

La recherche d'IIS-C que nous avons présentée pourrait être intégrée dans un programme de confection d'horaires selon le même fonctionnement de l'algorithme de recherche tabou présenté par Gamache et al. [52] (présenté à la page 188). Jusqu'à une certaine valeur de k fixée, appelée k_{max}, un horaire pour l'employé k est trouvé en résolvant le problème de plus court chemin $(RCSP_k)$. Puis, il est possible d'utiliser la recherche tabou pour déterminer si (IP_k) est réalisable. Si l'algorithme tabou exhibe une solution réalisable, alors cela signifie que l'horaire attribué à l'employé k est optimal. Sinon, si l'algorithme tabou n'est pas capable d'exhiber une solution à (IP_k) (i.e., l'output de tabou est "JE NE SAIS PAS"), alors notre méthode de recherche d'IIS peut chercher un IIS-C. Si l'IIS-C obtenu est vérifié par notre algorithme exact, alors nous avons la preuve que ce problème n'est pas réalisable. De plus, le gestionnaire en charge de la confection d'horaires peut déterminer où se trouve le problème et peut, selon l'IIS obtenu, imposer

des tâches-mères à certains employés en fonction de leur qualification. Si l'IIS-C obtenu n'est pas vérifié par l'algorithme exact ou si la recherche d'IIS échoue, alors il est possible de passer au deuxième échelon de la vérification faite par Gamache et al. [52] et de trouver un horaire pour k en résolvant le problème linéaire mixte (MIP_k). Puis, à nouveau, l'horaire obtenu peut être testé avec la recherche tabou et un IIS-C peut être recherché. À nouveau, si un IIS-C est obtenu et vérifié il est possible d'en tenir compte afin de modifier le problème et si l'IIS-C n'est pas vérifié ou si la recherche échoue, alors il est possible de résoudre le problème à nombres entiers (IP_k). Si $k > k_{max}$, le problème mixte (MIP_k) est résolu sans essayer de résoudre d'abord le problème de plus court chemin. Il faut tout de même avoir à l'esprit qu'il peut y avoir plusieurs IIS-C. Donc, afin de rendre le problème réalisable, il est fort probable qu'il faille répéter plusieurs fois le processus comprenant la recherche d'un IIS-C et la modification pour supprimer le conflit identifié par l'IIS-C.

DISCUSSION GÉNÉRALE ET CONCLUSION

Dans cette thèse, nous avons présenté trois développements concernant les problèmes de satisfaction de contraintes.

Premièrement, nous avons étudié le problème SAT. Nous avons présenté des algorithmes permettant de trouver des sous-ensembles incohérents irréductibles de clauses ou de variables pour ce problème. De plus, nous avons décrit un algorithme permettant d'extraire des IIS contenant un nombre minimum de clauses ou de variables. Nous avons aussi présenté des algorithmes permettant d'accélérer la recherche d'IIS ou de trouver des IIS plus denses ou de cardinalité plus petite. Nous avons décrit un algorithme de recherche tabou permettant de résoudre le problème Max-SAT pondéré. Cet algorithme tabou est utilisé par les algorithmes de recherche d'IIS. Nous avons effectué un grand nombre d'expérimentations afin de comparer nos résultats avec d'autres algorithmes effectuant cette recherche d'IIS. Ceci nous a permis de montrer que nos algorithmes trouvent la plupart du temps des IIS avec un nombre égal ou moindre de contraintes et de variables que les algorithmes avec lesquels nous nous sommes comparés. Par contre, nos algorithmes sont généralement plus lents que plusieurs autres algorithmes existants. Nous avons aussi créé des instances à partir de problèmes de k-coloration pour lesquelles seulement un de nos algorithmes a réussi à trouver des IIS de contraintes. Contrairement aux autres algorithmes effectuant la recherche d'IIS qui s'intéressent uniquement à l'extraction d'IIS de contraintes, nos méthodes peuvent être appliquées pour trouver des IIS de contraintes ou de variables. Nous avons vu que, pour accélérer les méthodes `Removal` et `Insertion`, il est intéressant de trouver d'abord des IIS de variables et ensuite des IIS de contraintes à partir des IIS-V trouvés précédemment. Nous avons montré que notre algorithme `HittingSet` est très efficace sur certaines instances pour trouver des IIS de cardinalité minimum.

Comme nous l'avons vu, les algorithmes de détection que nous avons proposés peuvent

être utilisés avec des algorithmes exacts (garantissant la non réalisabilité et la minimalité des IIS trouvés) ou avec des heuristiques (garantissant la minimalité dans certaines conditions bien précises). Pour la recherche d'IIS avec le premier algorithme proposé, `Removal`, nous avons utilisé un algorithme exact résolvant le problème SAT : `zChaff`. Pour les deux autres algorithmes proposés, nous avons utilisé un algorithme de recherche tabou résolvant le problème Max-SAT pondéré. Une variante qui pourrait être faite dans l'avenir consisterait à utiliser pour les algorithmes `Insertion` et `Hitting-Set` un algorithme exact résolvant le problème Max-SAT pondéré afin de voir si les temps de résolution obtenu peuvent être compétitifs avec ceux des autres algorithmes cherchant des IIS.

Dans la deuxième partie de la thèse, nous avons travaillé avec la contrainte globale SomeDifferent. Nous avons présenté un algorithme de filtrage cherchant à obtenir la cohérence de domaines pour cette contrainte. Cet algorithme combine un algorithme de recherche tabou qui essaie de trouver rapidement un support pour le plus de couples sommet-couleur possible avec un algorithme exact qui permet soit de filtrer, soit de valider les couples sommet-couleur restants. Nous avons testé notre algorithme sur plusieurs types d'instances : des instances aléatoires, des instances provenant d'un problème de planification de la main d'oeuvre, des instances ayant une unique D-coloration et des instances provenant de problèmes SUDOKU. Pour les deux premiers types d'instances, nous avons comparé nos résultats avec ceux de Richter et al. [122] : dans le cas des instances provenant d'un problème réel, les temps que nous avons obtenus pour effectuer le filtrage sont à peu près les mêmes que ceux de Richter et al.. Par contre, pour les instances aléatoires, notre algorithme est beaucoup plus rapide.

Un développement que nous prévoyons ajouter consiste à imbriquer notre algorithme de filtrage à l'intérieur d'un logiciel de programmation par contraintes. À cette fin, nous allons travailler avec un problème contenant d'autres types de contraintes en plus de la contrainte SomeDifferent. Nous voulons vérifier si le fait de maintenir la cohérence de

domaines pour la contrainte SomeDifferent (i.e., effectuer un premier filtrage au début de la résolution et propager les filtrages des autres contraintes pour la contrainte SomeDifferent en maintenant la cohérence de domaines) va nous permettre de gagner du temps pour la résolution du problème.

Les techniques mises en oeuvre ne sont pas spécifiques au problème de coloration de graphes. Elles pourraient être adaptées pour d'autres contraintes NP-difficiles. Il pourrait donc être intéressant de développer des algorithmes conçus selon la même technique pour d'autres contraintes NP-difficiles afin d'obtenir un algorithme de filtrage permettant d'obtenir la cohérence de domaine. Ceci peut être fait en combinant deux procédures spécifiques à la contrainte. Premièrement, une méthode de recherche locale cherche à trouver rapidement un support pour le plus de couples variable-valeur possible et deuxièmement, une méthode exacte peut filtrer ou valider les paires restantes.

Dans la troisième partie de la thèse, nous avons travaillé avec un problème de confection d'horaires pour le personnel navigant aérien. Nous avons présenté l'adaptation que nous avons faite pour ce problème particulier de deux des algorithmes de recherche d'IIS présentés dans la première partie, i.e., `Insertion` et `HittingSet`. Nous avons montré comment nous avons utilisé un algorithme de recherche tabou dans les algorithmes de recherche d'IIS. Nous avons aussi présenté un algorithme exact qui tient compte de toutes les contraintes du problème et qui permet de vérifier les IIS obtenus. Nous avons présenté différents résultats sur des instances classées en deux groupes. Le premier groupe contient différentes instances aléatoires et des instances ayant une unique D-coloration auxquelles nous avons ajouté une contrainte de qualification. Les instances aléatoires du premier groupe peuvent modéliser différents problèmes de confection d'horaires. Le deuxième groupe contient des instances aléatoires essayant de modéliser des problèmes de confection d'horaires pour le personnel navigant aérien. Pour toutes les instances nous avons cherché des IIS de contraintes seulement. Nous avons obtenu différents IIS de contraintes, ceux-ci étant parfois assez complexes, i.e., reliant des tâches et

des contraintes provenant de différentes tâches-mères et étant constitués de deux ou trois types de contraintes. Nous sommes satisfaits des résultats obtenus pour ce problème de confection d'horaires, car nous avons réussi à obtenir des IIS-C pour des instances simulant un problème réel dans des temps acceptables. De plus, certains des IIS-C obtenus ne sont pas évidents et n'auraient pas pu être détectés avec la méthode des compteurs. Grâce aux IIS que nous avons obtenus, nous avons montré que les méthodes de Galinier et Hertz [48] peuvent donc être appliquées efficacement à des CSP non binaires comportant des contraintes agissant sur plusieurs variables et des contraintes globales. La grande majorité des IIS obtenus ont pu être vérifiés avec l'algorithme exact dans des temps raisonnables. Nous avons aussi présenté comment nos algorithmes de recherche d'IIS peuvent être insérés dans un programme trouvant l'horaire de chaque employé selon l'ordre de séniorité.

Des améliorations pourraient être faites à nos algorithmes afin qu'ils tiennent compte d'autres contraintes existantes dans le problème de confection d'horaires. En effet, nous n'avons pas tenu compte par exemple des patrons de congé qui déterminent les congés obligatoires (nombre de jours consécutifs) que doit prendre un employé. De même, nous n'avons pas tenu compte des employés en réserve qui peuvent être considérés pour résoudre le problème.

RÉFÉRENCES

[1] Dimacs ftp site.
`ftp://dimacs.rutgers.edu/pub/challenge/sat/benchmarks/cnf`.
Dernière consultation : 22 janvier 2009.

[2] Sat benchmarks from automotive product configuration.
`http://www-sr.informatik.uni-tuebingen.de/~sinz/DC/`.
Dernière consultation : 22 janvier 2009.

[3] ACHOUR, H., GAMACHE, M., SOUMIS, F. et DESAULNIERS, G. (2007). An Exact Solution Approach for the Preferential Bidding System Problem in the Airline Industry. *Transportation Science*, vol.41, no.3, p.354–365.

[4] ALSINET, T., MANYÀ, F. et PLANES, J. (2008). An efficient solver for weighted Max-SAT. *Journal of Global Optimization*, vol.41, no.1, p.61–73.

[5] ALSINET, T., MANYÀ, F. et PLANES, J. (2003). Improved branch and bound algorithms for Max-SAT. *Proceedings of 6th International Conference on the Theory and Applications of Satisfiability Testing–SAT'2003* (S. Margherita Ligure - Portofino, Italie), p.408–415.

[6] AMALDI, E., PFETSCH, M. E. et TROTTER, JR., L. E. (1999). Some Structural and Algorithmic Properties of the Maximum Feasible Subsystem Problem. *Lecture Notes in Computer Science*, vol.1610, p.45–59.

[7] AMALDI, E., PFETSCH, M. E. et TROTTER, JR., L. E. (2003). On the maximum feasible subsystem problem, IISs and IIS-hypergraphs. *Mathematical Programming*, vol.95, no.3, p.533–554.

[8] APT, K. R. (2003). *Principles of Constraint Programming*. Cambridge University Press.

[9] ASAHIRO, Y., IWAMA, K. et MIYANO, E. (1996) Random Generation of Test Instanzes with Controlled Attributes. *Cliques, Coloring, and Satisfiability : The Second DIMACS Implementation Challenge* (Providence, USA), D.S.Johnson and M.A.Trick, Eds., vol. 26, p.377–394.

[10] ASIRELLI, P., SANTIS, M. D. et MARTELLI, M. (1985). Integrity constraints in logic databases. *Journal of Logic Programming*, vol.2, no.3 , p.221–232.

[11] BAILEY, J. et STUCKEY, P. J. (2005). Discovery of Minimal Unsatisfiable Subsets of Constraints Using Hitting Set Dualization. *Practical Aspects of Declarative Languages, 7th International Symposium–PADL'2005*, (Long Beach, CA, USA), M. V. Hermenegildo et D. Cabeza, Eds., *Lecture Notes in Computer Science*, vol.3350, Springer, p.174–186.

[12] BAKKER, R. R., DIKKER, F., TEMPELMAN, F. et WOGNUM, P. M. (1993). Diagnosing and Solving Over-Determined Constraint Satisfaction Problems. *Proceedings of the 13th International Joint Conference on Artificial Intelligence–IJCAI'93* (Chambéry, France), vol.1, Morgan Kaufmann, p.276–281.

[13] BARTÀK, R. On-line guide to constraint programming. `http://kti.mff.cuni.cz/~bartak/constraints/index.html`. Dernière consultation : 22 janvier 2009.

[14] BARTÀK, R. (1999). Constraint programming : In pursuit of the holy grail. *Proceedings of the Week of Doctoral Students (WDS99)*, Part IV, MatFyzPress, Prague, République Tchèque, p.555–564.

[15] BARTÀK, R. (2001). Theory and Practice of Constraint Propagation. *Proceedings of the 3rd Workshop on Constraint Programming for Decision and Control–CPDC'2001*, (Wydavnictvo Pracovni Komputerowej, Gliwice, Pologne), p.7–14.

[16] BATTITI, R. (1996). Reactive Search : Toward Self–Tuning Heuristics. *Modern Heuristic Search Methods*, V. J. Rayward-Smith, I. H. Osman, C. R. Reeves et G. D. Smith, Eds. John Wiley & Sons Ltd., Chichester, p.61–83.

[17] BATTITI, R. et PROTASI, M. (1998). Approximate Algorithms and Heuristics for MAX-SAT. *Handbook of combinatorial optimization*, D. Ding-Zhu et P. Pardalos, Eds., vol.1. Boston, Mass : Kluwer academic.

[18] BEAME, P., KARP, R., PITASSI, T. et SAKS, M. (2002). The efficiency of resolution and Davis-Putnam procedures. *SIAM Journal on Computing*, vol.31, Society for Industrial and Applied Mathematics, p.1048–1075.

[19] BERGE, C. (1989). *Graphes et Hypergraphes*. Dunod, Paris.

[20] BERGE, C. (1989). *Hypergraphs*, North Holland Mathematical Library, vol.45, Elsevier Science Publishers B.V (North-Holland).

[21] BIRO, M., HUJTER, M. et TUZA, Z. (1992). Precoloring extension. I. Interval graphs. *Discrete Math.*, vol.100, no.1-3, p.267–279.

[22] BONET, M. L., LEVY, J. et MANYÀ, F. (2007). Resolution for Max-SAT. *Artificial Intelligence*, vol.171, no.8-9, p.606–618.

[23] BORCHERS, B. et FURMAN, J. (1999). A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems. *Journal of Combinatorial Optimization*, vol.2, p.299–306.

[24] BOUSSEMART, F., HEMERY, F., LECOUTRE, C. et SAIS, L. (2004). Boosting systematic search by weighting constraints. *Proceedings of the Sixteenth European Conference on Artificial Intelligence–ECAI'04* (Valence, Espagne), R. L. de Mántaras et L. Saitta, Eds., IOS Press, p.146–150.

[25] BROWN, J. (1972). Chromatic Scheduling and the Chromatic Number Problem. *Management Science*, vol.19, no.4, p.456–463.

[26] BRUNI, R. (2003). Approximating minimal unsatisfiable subformulae by means of adaptative core search. *Discrete Applied Mathematics*, vol.130, no.2, p.85–100.

[27] BRUNI, R. (2005). On Exact Selection of Minimally Unsatisfiable Subformulae. *Annals of Mathematics and Artificial Intelligence*, vol.42, no.1-4, p.35–50.

[28] BUENING, H. K. et ZHAO, X. (2002). Polynomial time algorithms for computing a representation for minimal unsatisfiable formulas with fixed deficiency. *Information Processing Letters*, vol.84, p.147–151.

[29] BYRNE, J. (1988). A preferential bidding system for technical aircrew. *1988 AGIFORS Symposium Proceedings*, vol.28, p.87–99.

[30] CHEN, J. et KANJ, I. (2002). Improved Exact Algorithms for MAX-SAT. *LATIN '02 : Proceedings of the 5th Latin American Symposium on Theoretical Informatics* (Londre, Grande-Bretagne), Springer-Verlag, p.341–355.

[31] CHINNECK, J. W. (1997). Finding a Useful Subset of Constraints for Analysis in an Infeasible linear program. *INFORMS Journal on Computing*, vol.9, no.2, p.164–174.

[32] COOK, S. (1971). The complexity of theorem procedures. *Proceedings of the 3rd Annual ACM Symposium On Theory of Computing* (Shaker Heights, USA), ACM/IEEE, p.515–158.

[33] CRAWFORD, J. et AUTON, L. (1996). Experimental Results on the Crossover Point in Random 3-SAT. *Artificial Intelligence*, vol.81, p.31–57.

[34] CRAWFORD, J. M. et BAKER, A. B. (1994). Experimental Results on the Application of Satisfiability Algorithms to Scheduling Problems. *Proceedings of the Twelfth National Conference on Artificial Intelligence–AAAI'94)* (Seattle, USA), vol.2, American Association for Artificial Intelligence Press, p.1092–1097.

[35] CULBERSON, J.
`http://web.cs.ualberta.ca/\%7Ejoe/Coloring/index.html`.
Dernière consultation : 22 janvier 2009.

[36] DANTSIN, E., GAVRILOVICH, M., HIRSCH, E. et KONEV, B. (1998). Approximation algorithms for Max SAT : a better performance ratio at the cost of a longer running time. Rapport technique, PDMI Preprint 14/1998. Disponible à : `www.cs.ubc.ca/~hutter/previous-earg/EmpAlgReadingGroup/dantsin98approximation.pdf`.
Dernière consultation : 22 janvier 2009.

[37] DAVIS, M., LOGEMANN, G. et LOVELAND, D. (1962). A machine program for theorem-proving. *Communications of the ACM*, vol.5, no.7. p.394–397.

[38] DAVIS, M. et PUTNAM, H. (1960). A computing procedure for quantification theory. *Journal of the ACM*, vol.7, no.3, p.201–215.

[39] DESROSIERS, C., GALINIER, P. et HERTZ, A. (2008). Efficient algorithms for finding critical subgraphs. *Discrete Applied Math.*, vol.156, no.2, p.244–266.

[40] DESROSIERS, C., GALINIER, P., HERTZ, A. et PAROZ, S. (2008). Using heuristics to find minimal unsatisfiable subformulas in satisfiability problems. À paraître dans *Journal of Combinatorial Optimization.*

[41] EL IDRISSI, T. (2002). Amélioration de la méthode des compteurs pour la construction des blocs mensuels personnalisés d'agents de bord. Mémoire de maîtrise, Département de Mathématiques et Génie Industriel, École Polytechnique de Montréal.

[42] FEDERICI, F. et PASCHINA, D. (1985). Automated rostering model. *AGIFORS symposium proceedings*, vol.26, p.19–47.

[43] FLEISCHNER, H., KULLMANN, O. et SZEIDER, S. (2002). Polynomial-time Recognition of Minimal Unsatisfiable Formulas with Fixed Clause-variable Difference. *Theoretical Computer Science*, vol.289, no.1, p.503–516.

[44] FLEURENT, C. et FERLAND, J. (1996). Genetic and Hybrid Algorithms for Graph Coloring. *Annals of Operations Research*, vol.63, p.437–461.

[45] FRANCO, J., GU, J., PURDOM, P. W. et WAH, B. W. (1997). Satisfiability Problem : Theory and Applications. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, p.19–152.

[46] FREEMAN, J. (1995). Improvements to Propositional Satisfiability Search Algorithms. Thèse de doctorat, Department of Computer and Information Science, Univ. of Pennsylvania, Philadelphia.

[47] GALINIER, P. et HAO, J.-K. (1999). Hybrid evolutionary algorithms for graph coloring. *Journal of Combinatorial Optimization*, vol.3, no.4, p.379–397.

[48] GALINIER, P. et HERTZ, A. (2007). Solution techniques for the Large Set Covering Problem. *Discrete Applied Math.*, vol.155, no.3, p.312–326.

[49] GALINIER, P., HERTZ, A., PAROZ, S. et PESANT, G. (2008). Using Local Search to Speed Up Filtering Algorithms for Some NP-Hard Constraints. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR2008*, L. Perron et M. A. Trick, Eds., *Lecture Notes in Computer Science*, vol.5015, Springer, p.298–302.

[50] GALINIER, P., HERTZ, A., PAROZ, S. et PESANT, G. (2008). Using Local Search to Speed Up Filtering Algorithms for Some NP-Hard Constraints. Rapport technique G-2008-02, Les Cahiers du GERAD.

[51] GALLAIRE, H., MINKER, J. et NICOLAS, J.-M. (1984). Logic and Databases : A Deductive Approach. *ACM Computing Surveys*, vol.16, no.2, p.153–185.

[52] GAMACHE, M., HERTZ, A. et OUELLET, J. O. (2007). A graph coloring model for a feasibility problem in monthly crew scheduling with preferential bidding. *Computers and Operation Research*, vol.34, no.8, p.2384–2395.

[53] GAMACHE, M. et SOUMIS, F. (1998). A method for optimally solving the rostering problem. *Operations research in the arline industry*, Yu G. (Ed.), Kluwer, Norwell, MA, p.124–157.

[54] GAMACHE, M., SOUMIS, F., VILLENEUVE, D., DESROSIERS, J. et GELINAS, E. (1998). The Preferential Bidding System at Air Canada. *Transportation Science*, vol.32, no.3, p.246–255.

[55] GAREY, M. R. et JOHNSON, D. S. (1990). *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA.

[56] GENDREAU, M., HERTZ, A. et LAPORTE, G. (1994). A Tabu Search Heuristic for the Vehicle Routing Problem. *Management Science*, vol.40, p.1276–1290.

[57] GENT, I. et WALSH, T. (1999). The search for Satisfaction. Avant-projet d'article. Disponible à la page : www.cse.unsw.edu.au/ tw/sat/sat-survey2.pdf. Dernière consultation : 22 janvier 2009.

[58] GENT, I. P. et WALSH, T. (1993). Towards an Understanding of Hill-Climbing Procedures for SAT. *National Conference on Artificial Intelligence*, p.28–33.

[59] GERSHMAN, R., KOIFMAN, M. et STRICHMAN, O. (2006). Deriving Small Unsatisfiable Cores with Dominators. *Computer Aided Verification 2006*, T. Ball et R. B. Jones, Eds., *Lecture Notes in Computer Science*, Springer, vol.4144, p.109–122.

[60] GLANERT, W. (1984). A timetable approach to the assignement of pilots to rotations. *1984 AGIFORS symposium proceedings*, vol.24, p.369–391.

[61] GLEESON, J. et RYAN, J. (1990). Identifying Minimally Infeasible Subsystems of Inequalities. *ORSA Journal on Computing*, vol.2, no.1, p.61–63.

[62] GLOVER, F. et LAGUNA, M. (1997). *Tabu Search*. Kluwer Academic Publishers, Boston, USA.

[63] GRÉGOIRE, E., MAZURE, B. et PIETTE, C. (2005). A new local search algorithm to compute inconsistent kernels. *Proceedings of the 6th International Metaheuristics International Conference–MIC'05*, (Vienne, Autriche), actes électroniques (cd-rom).

[64] GRÉGOIRE, E., MAZURE, B. et PIETTE, C. (2006). Extracting MUSes. *Proceedings of the 17th European Conference on Artificial Intelligence–ECAI'06* (Trente, Italie), p.387–391.

[65] GRÉGOIRE, E., MAZURE, B. et PIETTE, C. (2006). Tracking MUSes and Strict Inconsistent Covers. *Proceedings of the 6th Conference on Formal Methods in Computer Aided Design–FMCAD'0* (San Jose, USA), p.39–46.

[66] GRÉGOIRE, E., MAZURE, B. et PIETTE, C. (2006). Une méta-heuristique basée sur le comptage de contraintes falsifiées. *Métaheuristiques–META'06*, Hammamet, Tunisia.

[67] GRÉGOIRE, E., MAZURE, B. et PIETTE, C. (2007). Boosting a Complete technique to Find MSS and MUS thanks to a Local Search Oracle. *Proceedings of the 20th International Joint Conference on Artificial Intelligence–IJCAI'07* (Hyderabad, India), vol.2, p.2300–2305.

[68] GRÉGOIRE, E., MAZURE, B., PIETTE, C. et LAKDHAR, S. (2006). A New Heuristic-based albeit Complete Method to Extract MUCs from Unsatisfiable CSPs. *Proceedings of the IEEE International Conference on Information Reuse and Integration–IEEE-IRI'06* (Waikoloa, USA), p.325–329.

[69] GRÉGOIRE, E., MAZURE, B. et PIETTE, C. (2007). Une nouvelle methode hybride pour calculer tous les MSS et tous les MUS. *Troisiemes Journees Francophones de Programmation par Contraintes–JFPC'07*, Rocquencourt, France.

[70] GU, J., PURDOM, P., FRANCO, J. et WAH, B. (1996). Algorithms for the Satisfiability (SAT) Problem : a Survey. *Satisfiability Problem : Theory and Applications*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, p.19–152.

[71] GUNOPULOS, D., KHARDON, R., MANNILA, H., SALUJA, S., TOIVONEN, H. et SHARMA, R. S. (2003). Discovering all most specific sentences. *ACM Transactions on Database Systems*, vol.28, no.2, p.140–174.

[72] HANSEN, P. et JAUMARD, B. (1990). Algorithms for the Maximum Satisfiability Problem. *Computing*, vol.44, no.4, p.279–303.

[73] HEMERY, F., LECOUTRE, C., SAÏS, L. et BOUSSEMART, F. (2006). Extracting MUCs from Constraint Networks. *17th European Conference on Artificial Intelligence–ECAI'06*, Trente, Italie, p.113–117.

[74] HERRMANN, F. et HERTZ, A. (2002). Finding the Chromatic Number by Means of Critical Graphs. *Journal of Experimental Algorithmics 7*, vol.7, no.10, p.1–9.

[75] HERTZ, A. et DE WERRA, D. (1987). Using tabu search for graph coloring. *Computing*, vol.39, p.345–351.

[76] HOOS, H. H. et STUTZLE, T. (2000). Local Search Algorithms for SAT : An Empirical Evaluation. *Journal of Automated Reasoning*, vol.24, no.4, p.421–481.

[77] HOPCROFT, J. et KARP, R. (1973). $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal of Computing*, vol.2, p.225–231.

[78] HUANG, J. (2005). MUP : a minimal unsatisfiability prover. *ASP-DAC '05 : Proceedings of the 2005 conference on Asia South Pacific design automation* (New York, USA), ACM, p.432–437.

[79] JANSEN, K. et SCHEFFLER, P. (1997). Generalized coloring for tree-like graphs. *Discrete Applied Mathematics*, vol.75, p.135–155.

[80] JEANDROZ, P. (2000). Heuristique pour la construction de blocs mensuels personnalisés d'agents de bord. Mémoire de maîtrise, Département de Mathématiques et Génie Industriel, École Polytechnique de Montréal.

[81] JEROSLOW, R. J. et WANG, J. (1990). Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, vol.1, p.167–188.

[82] JIANG, Y., KAUTZ, H. et SELMAN, B. (1995). Solving Problems with Hard and Soft Constraints Using a Stochastic Algorithm for MAX-SAT. *Proceedings of the 1st Workshop on Artificial Intellignce and Operations Research.*.

[83] JOHNSON, D. S. (1974). Approximation algorithms for combinatorial problems. *Journal of Computer and System Science*, vol.9, p.256–278.

[84] JOY, S., BORCHERS, B. et MITCHELL, J. E. (1997). A branch-and-cut algorithm for MAX-SAT and weighted MAX-SAT. *Satisfiability Problem : Theory and Applications*, vol. 35, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, p.519–536.

[85] KAMATH, A. P., KARMARKAR, N. K., RAMAKRISHNAN, K. G. et RESENDE, M. G. C. (1993). An interior point approach to Boolean vector function synthesis. *Proceedings of the 36th MSCAS*, p.185–189.

[86] KAUTZ, H. A. et SELMAN, B. (1992). Planning as Satisfiability. *Proceedings of the Tenth European Conference on Artificial Intelligence–ECAI'92* (Vienne, Autriche), p.359–363.

[87] KIRKPATRICK, S., GELATT, C. D. et VECCHI, M. P. (1983). Optimization by Simulated Annealing. *Science*, vol.220, no.4598, p.671–680.

[88] KUBALE, M. et JACKOWSKI, B. (1985). A generalized implicit enumeration algorithm for graph coloring. *Communications of the ACM*, vol.28, no.4, p.412–418.

[89] KUNZ, W., MARQUES-SILVA, J. et MALIK, S. (2002). SAT and ATPG : algorithms for Boolean decision problems. Logic Synthesis and Verification, Kluwer Academic Publishers, p.309–341.

[90] LARRABEE, T. (1992). Test Pattern Generation Using Boolean Satisfiability. *IEEE Transactions on Computer-Aided Design*, vol.11, no.1, p.6–22.

[91] LEWIN, M., LIVNAT, D. et ZWICK, U. (2002). Improved rounding techniques for the MAX 2-SAT and MAX DI-CUT problems. *Integer Programming and Combinatorial Optimization (IPCO), LNCS* , vol.2337, p.67–82.

[92] LI, C. M. et ANBULAGAN (1997). Heuristics Based on Unit Propagation for Satisfiability Problems. *Proceedings of 15th International Joint Conference on Artificial Intelligence*, p.366–371.

[93] LI, C. M., MANYÀ, F. et PLANES, J. (2006). Detecting disjoint inconsistent subformulas for computing lower bounds for max-sat. *Proceedings of the 21st National Conference on Artificial Intelligence–AAAI'2006,* (Boston, USA), p.86–91.

[94] LI, C. M., MANYÀ, F. et PLANES, J. (2007). New Inference Rules for Max-SAT. *Journal of Artificial Intelligence Research*, vol.30, p.321–359.

[95] LIFFITON, M. H. et SAKALLAH, K. A. (2005). On Finding All Minimally Unsatisfiable Subformulas. *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing–SAT'2005)* (St. Andrews,

Écosse), F. Bacchus et T. Walsh, Eds., *Lecture Notes in Computer Science*, Springer, vol.3569, p.173–186.

[96] LIFFITON, M. H. et SAKALLAH, K. A. (2008). Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints. *Journal of Automated Reasoning*, vol.40, no.1, p.1–33.

[97] LOUGHRY, J., VAN HEMERT, J. et SCHOOFS, L. (2000). Efficiently Enumerating the Subsets of a Set.
Disponible à la page : www.applied-math.org/subset.pdf.
Dernière consultation : 22 janvier 2009.

[98] LYNCE, I. et MARQUES-SILVA, J. (2004). On Computing Minimum Unsatisfiable Cores. *Proceedings of The Seventh International Conference on Theory and Applications of Satisfiability Testing–SAT'2004* (Vancouver, Canada), p.305–310.

[99] MAHDIAN, M. et MAHMOODIAN, E. S. (1999). A Characterization of Uniquely 2-list Colorable Graphs. *Ars Combinatoria*, vol.51, p.295–305.

[100] MARQUES-SILVA, J. et SAKALLAH, K. A. (1999). GRASP : A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, vol.48, no.5, p.506–521.

[101] MAZURE, B., GRÉGOIRE, E. et SAÏS, L. (1998). Boosting complete techniques thanks to local search methods. *Annals of Mathematics and Artificial Intelligence*, vol.22, p.319–331.

[102] MAZURE, B., SAÏS, L. et GRÉGOIRE, E. (1997). Tabu search for SAT. *Proceedings of the Fourteenth National Conference on Artificial Intelligence–AAAI'97* (Providence, USA), p.281–285.

[103] MCALLESTER, D., SELMAN, B. et KAUTZ, H. (1997). Evidence for Invariants in Local Search. *Proceedings of the Fourteenth National Conference on Artificial Intelligence–AAAI'97* (Providence, Rhode Island), p.321–326.

[104] MEHROTRA, A. et TRICK, M. A. (1996). A column generation approach for graph coloring. *INFORMS Journal on Computing*, vol.8, p.344–354.

[105] MLADENOVIĆ, N. et HANSEN, P. (1997). Variable neighborhood search. *Computers and Operations Research*, vol.24, no.11, p.1097–1100.

[106] MNEIMNEH, M. N., LYNCE, I., ANDRAUS, Z. S., MARQUES-SILVA, J. et SAKALLAH, K. A. (2005). A Branch and Bound Algorithm for Extracting Smallest Minimal Unsatisfiable Formulas. *International Conference on Theory and Applications of Satisfiability Testing*, F. Bacchus et T. Walsh, Eds., vol.3569, p.467–474.

[107] MOSKEWICZ, M. W., MADIGAN, C. F., ZHAO, Y., ZHANG, L. et MALIK, S. (2001). Chaff : Engineering an Efficient SAT Solver. *Proceedings of the 38th Design Automation Conference–DAC'01* (Las Vegas, USA), p.530–535.

[108] NGUYEN, T., PERKINS, W., LAFFEY, T. et PECORA, D. (1985). Checking an expert system knowledge base for consistency and completeness. *Proceedings of the 9th International Joint Conference on Artificial Intelligence–IJCAI'85* (Los Altos, USA), vol.1, p.375–378.

[109] OH, Y., MNEIMNEH, M. N., ANDRAUS, Z. S., SAKALLAH, K. A. et MARKOV, I. L. (2004). AMUSE : A Minimally-Unsatisfiable Subformula Extractor. *Proceedings of the of the 41th Design Automation Conference–DAC'2004)* (San Diego, USA), ACM/IEEE, p.518–523.

[110] OUELLET, J. (2004). Une approche tabou pour le problème d'horaires de personnel en transport aérien. Mémoire de maîtrise, Département de Mathématiques et Génie Industriel, École Polytechnique de Montréal.

[111] PAPADIMITRIOU, C. H. et WOLFE, D. (1988). The Complexity of Facets Resolved. *Journal of Computer and System Sciences*, vol.37, p.2–13.

[112] PAPADIMITRIOU, C. H. et YANNAKAKIS, M. (1982). The Complexity of Facets (and Some Facets of Complexity). *Proceedings of the fourteenth annual ACM symposium on Theory of computing* (San Diego, USA), p.255–260.

[113] PATURI, R., PUDLÁK, P., SAKS, M. E. et ZANE, F. (1998). An improve exponentiel-time algorithm for k-SAT. *Proceedings of the 39th Annual Symposium on Foundations of Computer Science–FOCS'98* (Washington, USA), p.628–637.

[114] PATURI, R., PUDLÁK, P., SAKS, M. E. et ZANE, F. (2005). An improve exponentiel-time algorithm for k-SAT. *Journal of the ACM*, vol.52, no.3, p.337-364.

[115] PEEMOELLER, J. (1983). A Correction to Brélaz's Modification of Brown's Coloring Algorithm. *Communications of the ACM*, vol.26, no.8, p.593–597.

[116] PESANT, G. (2001). A Filtering Algorithm for the Stretch Constraint. *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming–CP'2001* (Paphos, Chypre), T. Walsh, Ed., p.183–195.

[117] R. MOORE, J. EVANS, H. N. (1978). Computerized tailored blocking. *1978 AGIFORS Symposium Proceedings*, vol.18, p.343–361.

[118] RÉGIN, J.-C. (1999). Arc consistency for global cardinality with costs. *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming–CP'99* (Alexandria, USA), p.390–404.

[119] RÉGIN, J.-C. (1994). A filtering algorithm for constraints of difference in CSPs. *Proceedings of the twelfth national conference on Artificial intelligence–AAAI'94* (Menlo Park, USA), American Association for Artificial Intelligence, vol.1, p.362–367.

[120] RÉGIN, J.-C. (1996). Generalized Arc Consistency for Global Cardinality Constraint. *Proceedings of the fourteenth national conference on Artificial Intelligence–AAAI'96* (Portland, USA), p.209–215.

[121] RESENDE, M. G. C. et FEO, T. A. (1996). A GRASP for satisfiability. *Cliques, Coloring and Satisfiability : Second DIMACS Implementation Challenge, DIMACS Series In Discrete Mathematics and Theoretical Computer Science*, D. S. Johnson et M. A. Trick, Eds., vol.26. AMS, p. 99–520.

[122] RICHTER, Y., FREUND, A. et NAVEH, Y. (2006). Generalizing AllDifferent : The SomeDifferent Constraint. *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming–CP'06* (Nantes, France), vol.4204/2006, Springer, p.468–483.

[123] SAROIU, S. (1999). An Overview of the MOMS Heuristics. http ://research.microsoft.com/users/ssaroiu/publications/classes/cse573/sat.ps. Dernière consultation : 22 janvier 2009.

[124] SARRA, D. (1988). The automatic assignment model. *1988 AGIFORS symposium proceedings*, vol.28, p.23–37.

[125] SCHMIDT, W. et HOSSEINI, J. (1994). Preferential schedule assignments for airline crew scheduling. *Proceedings of the Joint National Conference of the Operations Research Society of America and The Institue of Management Sciences conference (ORSA/TIMS conference)*, (Detroit, USA).

[126] SELMAN, B. et KAUTZ, H. A. (1993). Domain-independant Extensions to GSAT : Solving Large Structured Variables. *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence–IJCAI'93*, (Chambéry, France), p.290–295.

[127] SELMAN, B. et KAUTZ, H. A. (1993). An Empirical Study of Greedy Local Search for Satisfiability Testing. *Proceedings of the Eleventh National Conference on Artificial Intelligence–AAAI'93* (Washington, USA), p.46–51.

[128] SELMAN, B., KAUTZ, H. A. et COHEN, B. (1994). Noise strategies for improving local search. *Proceedings of the Twelfth National Conference on Artificial Intelligence–AAAI'94* (Seattle, USA), p.337–343.

[129] SELMAN, B., LEVESQUE, H. et MITCHELL, D. (1992). A new method for solving hard satisfiability problems. *Proceedings of the Tenth National Conference on Artificial Intelligence–AAAI'92*, (San Jose, USA), p.440–446.

[130] SHERALI, H. et SOYSTER, A. (1983). Preemptive and non preemptive multi-objective programming : relationships and counter. *Journal of Optimisation Theory and Applications*, vol.39, p.173–186.

[131] STÜTZLE, T., HOOS, H. et ROLI, A. (2001). A review of the literature on local search algorithms for MAX-SAT. Rapport technique AIDA–01–02, Intellectics Group, TU Darmstadt, Allemagne.

[132] TARJAN, R. (1972). Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, vol.1, p.146–160.

[133] TINGLEY, G. (1979). Still another solution method for the monthly aircrew assignement problem. *1979 AGIFORS symposium proceedings*, vol.19, p.143–203.

[134] TRICK, M.
`http://mat.gsia.cmu.edu/COLOR/solvers/trick.c`.
Dernière consultation : 22 janvier 2009.

[135] TSANG, E.P.K.(1993) *Foundations of Constraint Satisfaction*. Academic Press, Londres et San Diego.

[136] VAN HOEVE, W.-J. (2001). The all_different Constraint : A Survey. *Proceedings of the Sixth Annual Workshop of the ERCIM Working Group on Constraints*, Prague, Rép. Tchèque.

[137] WALSH, T. (2000). SAT v CSP. *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming-CP'2000* (Singapour), vol.1894, Springer-Verlag, p.441–456.

[138] ZHANG, H. (1997). Specifying latin square problems in propositional logic. *Automated Reasoning and Its Applications : essays in honor of Larry Wos*, p. 115-146.

[139] ZHANG, H. (1997). SATO : An Efficient Propositional Prover. *Proceedings of International Conference on Automated Deduction–CADE'97)* (Townsville, Australie), p.272–275.

[140] ZHANG, L. et MALIK, S. (2003). Extracting small unsatisfiable cores from unsatisfiable boolean formulas. *Sixth International Conference on Theory and Applications of Satisfiability Testing–SAT'2003* (S. Margherita Ligure - Portofino, Italie), p.518–523.

ANNEXE I

MÉTHODES DE RECHERCHE LOCALE

La plupart des problèmes que nous traitons dans cette thèse sont NP-difficiles ou NP-complets. C'est pour cette raison que nous avons développé plusieurs algorithmes utilisant des méthodes de recherche locale (plus précisément des algorithmes de recherche tabou). Dans cette annexe, nous présentons d'abord de façon générale les algorithmes de recherche locale, puis nous détaillons la méthode de descente et la recherche tabou.

Considérons un problème pour lequel l'espace des solutions est noté $\mathcal{S}$. Soit $s \in \mathcal{S}$ une configuration ou solution de l'espace de recherche. On note $f(s)$ la valeur de la fonction objectif. Le problème consiste à minimiser la valeur de la fonction f. Un **algorithme de recherche locale** choisit une solution initiale $s_0 \in \mathcal{S}$, ensuite elle génère des solutions $s_1, s_2, ...$ dans $\mathcal{S}$ de telle sorte que $s_{i+1} \in N(s_i)$, où $N(s) \subset \mathcal{S}$ est appelé *le voisinage* de s : c'est-à-dire qu'il est possible d'obtenir une solution $s' \in N(s)$ à partir de s en faisant de petits changements (mouvements) en respectant certaines règles prédéfinies :

$$N(s) = \{s' \in \mathcal{S} : s' \text{ est un voisin de } s\}.$$

Les principales méthodes de recherche locale sont : la méthode de descente, la recherche tabou [62], la méthode du recuit simulé [87] et la méthode de recherche à voisinages variables [105]. Ci-dessous sont exposées deux méthodes de recherche locale : la méthode de descente qui est la méthode de recherche locale la plus simple et la méthode tabou que nous utiliserons dans la thèse.
Pour chaque méthode de recherche locale, il faut définir les éléments suivants :

- l'espace des solutions $\mathcal{S}$,

- la fonction objectif f qui sert à évaluer la valeur d'une solution $s \in \mathcal{S}$
- le voisinage $N(s)$ d'une solution $s \in \mathcal{S}$,
- la manière de choisir une solution dans $N(s)$, c'est-à-dire les mouvements autorisés,
- un critère d'arrêt.

I.1 Algorithme de descente

Nous allons d'abord présenter l'algorithme général de descente [62]. L'algorithme de descente choisit à chaque étape une solution voisine meilleure que la solution courante. Il existe plusieurs versions de l'algorithme de descente : par exemple de première amélioration (*first improvement*) qui choisit la première solution voisine qui améliore la solution courante, de dernière amélioration (*last improvement*) qui choisit la dernière solution voisine qui améliore la solution courante ou de meilleure amélioration (*best improvement*) qui choisit la meilleure solution voisine. Le pseudo-code d'un algorithme de descente de meilleure amélioration est donné dans la Figure I.1. L'algorithme s'arrête dès qu'il ne peut plus améliorer la solution. L'algorithme s'arrête donc dans le premier minimum local rencontré (i.e. pour une solution s tel que $f(s') \geq f(s) \ \forall \ s' \in N(s)$), la valeur de ce dernier pouvant être beaucoup plus grande que celle du minimum global. On peut voir ce défaut dans la Figure I.2 : l'algorithme de descente commence en s_0. Il visite successivement les solutions s_1 et s_2 avant de s'arrêter en s_3 qui est un minimum local. Dans cet exemple, le minimum global est s^*.

Pour remédier à ce genre d'inconvénient on peut utiliser la méthode tabou.

Algorithme de descente

construire une solution initiale quelconque s de $\mathcal{S}$;
poser $t \leftarrow s$;
poser $OK \leftarrow vrai$;
tant que OK *est* $vrai$ **faire**
  déterminer $s' \in N(t)$ tq $f(s') \leq f(s'') \ \forall\, s'' \in N(t)$;
  si $f(s') \geq f(t)$ **alors** $OK \leftarrow faux$;
  sinon si $f(s') < f(t)$ **alors**
    $t \leftarrow s'$;

Figure I.1 – Algorithme de descente

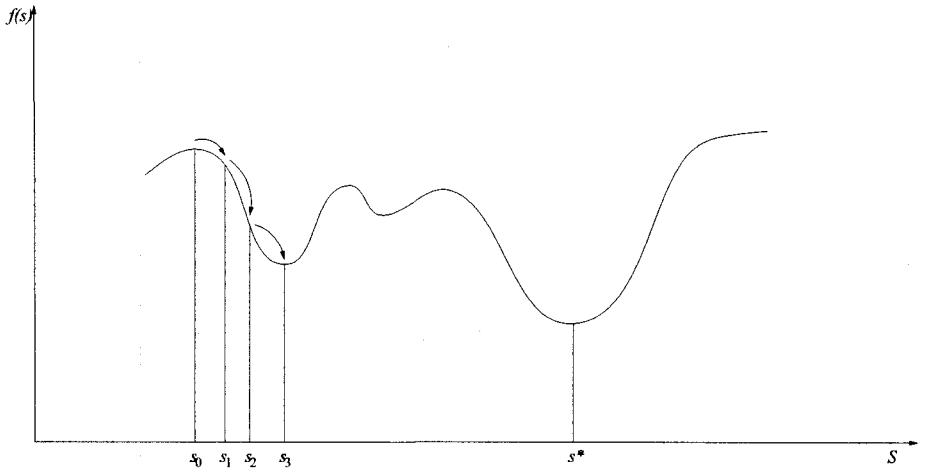


Figure I.2 – Exemple de blocage dans un minimum local

I.2 Algorithme tabou

La méthode tabou [62] est aussi itérative : elle consiste à générer à chaque étape un sous-ensemble de solutions et à retenir la meilleure solution parmi elles. Elle fait appel à un ensemble de règles permettant de guider la recherche dans l'espace des solutions admissibles S.

Définition I.2.1. Une *liste tabou* est une mémoire flexible, à court terme, qui garde une trace des opérations passées. Nous allons noter T une liste tabou.

Pour éviter de cycler, il faut se rappeler des derniers mouvements effectués ou des dernières solutions visitées. Il y a en effet deux manières de gérer une liste tabou T.

1. Des solutions sont stockées dans T. Cette manière de faire a pour principal désavantage d'utiliser une très grande place mémoire. Les solutions appartenant à T sont interdites pendant une durée déterminée (qui est précisée ci-dessous).
2. L'inverse des mouvements effectués est stocké dans T. Cette façon de faire a pour avantage d'être rapide et d'utiliser peu de place mémoire. Son désavantage est qu'il est possible qu'un mouvement interdit permette d'aboutir à une solution non encore rencontrée. Afin de pouvoir quand même visiter des solutions taboues dans des cas bien déterminés, il est possible de créer une fonction dite d'aspiration déterminant quand le statut tabou peut être levé. Par exemple, un mouvement tabou peut être effectué si celui-ci améliore la meilleure solution rencontrée. Les mouvements appartenant à T sont interdits pendant une durée déterminée.

La durée de l'interdiction est appelée **longueur tabou**. C'est un paramètre important à déterminer. En effet, si la longueur de la liste tabou est fixée à une valeur trop élevée, l'algorithme risque de ne pas avoir accès à des solutions non encore visitées. De même, si on fixe une valeur pour la longueur trop faible, alors la méthode risque d'être bloquée

dans un optimum local.

Un mouvement tabou ne peut pas être effectué, à moins que le statut tabou de ce mouvement soit levé par la fonction d'aspiration. Si la fonction d'aspiration lève le statut tabou des solutions améliorant la meilleure solution rencontrée, alors nous définirons $N'(s)$ comme l'union de l'ensemble des solutions s' obtenues à partir de s en effectuant des mouvements non tabou et des solutions $s' \in N(s)$ vérifiant $f(s') < f(s^*)$ (s^* étant la meilleure solution rencontrée). Ce deuxième cas (solutions s' telles que $f(s') < f(s^*)$) est appelé le critère d'aspiration. Il permet de lever le statut tabou d'une solution s' dans le cas où cette solution est meilleure que toutes les solutions rencontrées jusque-là. L'algorithme tabou choisit à chaque itération la solution voisine s' telle que $f(s') = arg\min_{s'' \in N'(s)} f(s'')$.

Contrairement à l'algorithme de descente, la méthode tabou ne s'arrête pas nécessairement si $f(s') \geq f(s)$. Le meilleur voisin de s n'entraîne donc pas obligatoirement une diminution de la fonction objectif f. Lorsque la méthode choisit une solution moins bonne que la solution courante, elle se dirige alors vers la solution voisine qui entraîne le plus faible accroissement possible de la fonction objectif. Cette dégradation de f devrait permettre à l'algorithme d'atteindre ultérieurement des régions de l'espace de recherche $\mathcal{S}$ contenant des solutions meilleures que celles déjà rencontrées.

La procédure est décrite dans la Figure I.3. Pour stopper la recherche on utilise un critère d'arrêt. Celui-ci peut être :

- un nombre maximum d'itérations,
- un nombre maximum d'itérations sans amélioration de s^*,
- un temps fixé de CPU,
- lorsque $N(s) - T = \emptyset$.

La méthode tabou est, en général, beaucoup plus performante qu'une méthode de descente mais également plus coûteuse en termes de ressource informatique (temps de calcul plus long, utilisation de plus de mémoire). L'espace de recherche n'est pas forcément entièrement visité et donc l'optimum global n'est pas toujours visité.

Algorithme Tabou

Initialisation
choisir une solution initiale $s \in \mathcal{S}$;
poser $s^* \leftarrow s$ et $f^* \leftarrow f(s)$;
poser $T \leftarrow \emptyset$;

Amélioration
tant que *aucun critère d'arrêt n'est satisfait* **faire**
déterminer $N'(s) \leftarrow \{s' \in N(s) \text{ tq } s' \notin T \text{ ou } f(s') < f^*\}$, c'est-à-dire
générer le voisinage de s au moyen de mouvements non tabou ou de
mouvements tabou qui aboutissent à des solutions qui améliorent s^*;
déterminer s' tq $f(s') \leftarrow argmin_{s'' \in N'(s)} f(s'')$;
si $f(s') < f^*$ **alors** poser $f^* \leftarrow f(s')$ et $s^* \leftarrow s'$;
poser $s \leftarrow s'$;
mettre à jour T;

retourner s^*

Figure I.3 – Algorithme tabou